



MIMER/SQL

Programmer's Manual

Version 7.3

Copyright © 1996 Sysdeco Mimer AB

MIMER/SQL version 7.3 Programmer's Manual

November, 1996

Copyright © 1996 Sysdeco Mimer AB.

Published by Sysdeco Mimer AB,

P.O.Box 1713,

S-751 47 Uppsala, Sweden.

Tel +46(0)18-18 50 00.

Fax +46(0)18-18 51 00.

Internet: <http://www.mimer.se>

Produced by Sysdeco Mimer AB, Uppsala, Sweden.

All rights reserved under international copyright conventions.

The contents of this manual may be printed in limited quantities for use at a Mimer installation site. No parts of the manual may be reproduced for sale to a third party.

FOREWORD

This manual describes how SQL statements may be embedded in application programs written in conventional host languages.

Intended audience

The manual is intended for application programmers working with MIMER/SQL.

Prerequisites

Application programmers using this manual are assumed to have a working acquaintance with the principles of relational database management in general and of MIMER in particular.

It is also assumed that programmers writing embedded SQL applications are familiar with the principles of the SQL database management language. Knowledge of MIMER/SQL is of course an advantage, although experience with other standard-compliant SQL implementations will suffice. Experience of MIMER/SQL is best gained by using the Interactive SQL facility.

Competence in at least one of the host programming languages supporting embedded MIMER/SQL (i.e. C, COBOL or FORTRAN) is assumed.

Organization of this manual

The organization of the material in this manual reflects the general requirements for writing application programs with ESQL:

- Chapter 1** introduces Embedded SQL (ESQL) in relation to the other MIMER/SQL products.
- Chapter 2** presents the basic principles of embedding SQL in application programs and describes features of preprocessing and compiling ESQL programs.
- Chapter 3** describes the way in which ESQL communicates with the host program, through host variables and dedicated communication areas.
- Chapter 4** describes how to log in to the database from application programs and how to make use of program identfs.
- Chapter 5** describes how to access data in the database tables; how to retrieve data and how to change table contents.
- Chapter 6** describes the essential features of transaction handling.
- Chapter 7** describes the special features of dynamic SQL, which allow an application program to process SQL statements entered by the user at run-time.
- Chapter 8** describes how to handle errors and exception conditions.
- Appendix A-F** describe host language dependent aspects of ESQL and preprocessors, SQL return codes and program examples.

Related MIMER publications

- **MIMER/SQL Programmer's Manual** contains a description of how MIMER/SQL can be embedded within application programs, written in conventional programming languages.
- **MIMER/SQL Interactive User's Manual** contains a description of the Interactive SQL and Batch SQL facilities. A user-oriented guide to the SQL statements is also included, which may provide help for less experienced users in formulating statements correctly (particularly the SELECT statement, which can be quite complex).
- **MIMER Shadowing Reference Manual** describes Shadowing, which allows several concurrent copies of a databank to be updated at the same time. This provides extra resilience to disk crashes and allows "backup on the fly".
- **MIMER System Management Handbook** describes system administration functions, including export/import, backup/restore, and the statistics functionality. The information in this manual is used primarily by the system administrator, and is not required by application program developers.

- **Machine-dependent information** comprises an Installation Guide and a Users Guide for each machine environment where MIMER is available. The Installation Guide is required only by the system administrator. The Users Guide contains machine-specific information relevant to the use of MIMER in the environment concerned, and should be available to all users.
- **Other MIMER manuals** are required only when other MIMER modules are used.

Suggestions for further reading

We can recommend to users of MIMER/SQL the many works of C. J. Date. His insight into the potential and limitations of SQL, coupled with his pedagogical talents, makes his books invaluable sources of study material in the field of SQL theory and usage. In particular, we can mention:

The Guide to the SQL Standard (Third Edition, 1994). ISBN: 0-201-55822-X. This work contains much constructive criticism and discussion of the SQL standard, including “SQL2”.

Official documentation of the accepted SQL standards may be found in:

ISO/IEC 9075:1989 Database Language SQL with integrity enhancement. This document describes the standard referred to as SQL1.

ISO/IEC 9075:1992(E) Information technology - Database languages - SQL. This document contains the standard referred to as SQL2.

X/Open Portability Guide, issue 3, volume 5, Data Management. ISBN: 0136858767. This document contains the specification referred to as XPG3 SQL.

CAE specification, Structured Query Language (SQL). X/Open document number: C201. ISBN: 1 872630 58 8. This document contains the X/Open-92-SQL specification.

CAE specification, Data Management: Structured Query Language (SQL), Version 2. X/Open document number: C449. ISBN: 1-85912-151-9.

Acronyms and trademarks

IEC	International Electrotechnical Commission
ISO	International Standards Organization
SQL	Structured Query Language
X/Open	X/Open is a trademark of the X/Open Company

CONTENTS

1	INTRODUCTION	
2	EMBEDDING MIMER/SQL IN APPLICATION PROGRAMS	
2.1	The scope of embedded MIMER/SQL	2-1
2.2	General principles for embedding SQL statements	2-2
2.2.1	Host languages	2-2
2.2.2	Identifying SQL statements	2-2
2.2.3	Comments	2-3
2.2.4	Recommendations	2-3
2.3	Processing embedded SQL	2-3
2.3.1	Preprocessing	2-3
2.3.2	Processing embedded SQL - the compiler	2-4
2.4	Essential program structure	2-4
3	COMMUNICATING WITH THE APPLICATION PROGRAM	
3.1	Host variable usage	3-1
3.1.1	Declaring host variables	3-1
3.1.2	Using variables in statements	3-2
3.1.3	Indicator variables	3-2
3.2	The SQLSTATE variable	3-3
3.3	The diagnostics area	3-4
3.4	The SQL Descriptor Area	3-4
4	IDENTS AND DATABASE CONNECTIONS	
4.1	MIMER idents	4-1
4.2	Access to the database	4-3
4.3	Connecting to a database	4-4
4.3.1	Connecting	4-4
4.3.2	Changing connection	4-5
4.3.3	Disconnecting	4-5
4.3.4	Program idents – ENTER and LEAVE	4-6
5	ACCESSING DATABASE TABLES	
5.1	Retrieving data	5-1
5.1.1	Declaring host variables	5-1
5.1.2	Declaring the cursor	5-2
5.1.3	Opening the cursor	5-2
5.1.4	Retrieving data	5-3
5.1.5	Closing the cursor	5-4
5.2	Retrieving single rows	5-5
5.3	Retrieving data from multiple tables	5-5
5.3.1	The 'Parts explosion' problem	5-6
5.4	Entering data into tables	5-8
5.4.1	Cursor-independent operations	5-8
5.4.2	Updating and deleting through cursors	5-8

6	TRANSACTION HANDLING AND DATABASE SECURITY	
6.1	Transaction principles	6-1
6.1.1	Concurrency control	6-1
6.1.2	Locking.....	6-3
6.2	Transaction control statements.....	6-4
6.2.1	Starting transactions	6-4
6.2.2	Ending transactions	6-5
6.2.3	Read-through-write-set	6-6
6.2.4	Statements allowed in transactions.....	6-7
6.2.5	Cursors in transactions.....	6-8
6.2.6	Error handling in transactions	6-9
6.3	Transactions and logging.....	6-9
6.4	Protection against data loss.....	6-10
6.4.1	System interruptions.....	6-10
6.4.2	Hardware failure.....	6-10
7	DYNAMIC SQL	
7.1	Principles of dynamic SQL	7-1
7.2	General summary of dynamic SQL processing.....	7-3
7.3	SQL descriptor area.....	7-4
7.3.1	The structure of the SQL descriptor area.....	7-4
7.3.2	The item descriptor area fields	7-5
7.3.3	SQL data type codes.....	7-8
7.4	Preparing statements	7-10
7.5	Extended dynamic cursors.....	7-11
7.6	Describing prepared statements	7-12
7.6.1	Describing SELECT-lists.....	7-13
7.6.2	Describing input variables	7-13
7.7	Executing statements	7-14
7.7.1	Non-SELECT statements.....	7-14
7.7.2	SELECT statements.....	7-15
7.8	Example framework for dynamic SQL programs	7-16
8	HANDLING ERRORS AND EXCEPTION CONDITIONS	
8.1	Syntax errors.....	8-1
8.2	Semantic errors	8-1
8.3	Run-time errors.....	8-2
8.3.1	GET DIAGNOSTICS.....	8-3
8.3.2	Testing for run-time errors and exception conditions	8-6
A	HOST LANGUAGE DEPENDENT ASPECTS	
A.1	Embedded SQL in C programs.....	A-2
A.1.1	SQL statement format	A-2
A.1.2	Included code.....	A-3
A.1.3	SQLCA declaration.....	A-3
A.1.4	Host variables	A-3
A.1.5	Preprocessor output format	A-5
A.1.6	Scope rules.....	A-5
A.2	Embedded SQL in COBOL programs	A-6
A.2.1	SQL statement format	A-6
A.2.2	Included code.....	A-7
A.2.3	Restrictions	A-8
A.2.4	SQLCA declaration.....	A-8
A.2.5	Host variables	A-8
A.2.6	Preprocessor output format	A-11
A.2.7	Scope rules.....	A-11

A.3	Embedded SQL in FORTRAN programs	A-13
A.3.1	SQL statement format	A-13
A.3.2	Included code	A-14
A.3.3	Restrictions	A-14
A.3.4	SQLCA declaration.....	A-14
A.3.5	Host variables	A-15
A.3.6	Preprocessor output format	A-17
A.3.7	Scope rules.....	A-17
B	PREPROCESSING APPLICATION PROGRAM SOURCE CODE	
B.1	Input and output files	B-1
B.2	Preprocessor options	B-2
B.2.1	C.....	B-2
B.2.2	COBOL	B-2
B.2.3	FORTRAN.....	B-2
C	RETURN CODES	
C.1	SQLSTATE return codes	C-2
C.2	Internal MIMER return codes.....	C-4
C.3	Warnings and messages.....	C-4
C.4	ODBC errors.....	C-5
C.5	Data-dependent errors.....	C-7
C.6	Limits exceeded	C-8
C.7	SQL statement errors.....	C-9
C.8	Program-dependent errors.....	C-15
C.9	Databank and table errors	C-17
C.10	Miscellaneous errors	C-19
C.11	Internal errors	C-22
D	DEPRECATED FEATURES	
D.1	INCLUDE SQLCA.....	D-1
D.2	SQLCODE	D-1
D.3	SQLDA	D-1
D.4	Parameter marker representation.....	D-2
D.5	VARCHAR(size)	D-2
E	PHYSICAL IMPLEMENTATION	
E.1	Databank handling.....	E-1
E.1.1	Databank-id's	E-1
E.1.2	Databank files	E-1
E.2	Table storage structure.....	E-2
E.3	Views.....	E-3
E.4	Indexes.....	E-3
E.5	Compiled statements	E-3
E.6	Work space requirements.....	E-4
F	APPLICATION PROGRAM EXAMPLES	
F.1	C program example.....	F-3
F.2	COBOL program example.....	F-5
F.3	FORTRAN program example	F-7
F.4	Dynamic SQL program example	F-9
F.5	Dynamic SQL example for handling binary data.....	F-29

1 INTRODUCTION

MIMER is an advanced relational database management system developed by Sysdeco Mimer AB. In version 7.3, the database management language MIMER/SQL is compatible in all essential features with the established SQL standards (see the MIMER/SQL Reference Manual for details).

MIMER/SQL is accessed through four user interfaces:

- Embedded SQL (ESQL) is used through a host programming language (C, COBOL or FORTRAN as available on the host computer). SQL statements are included as part of the source code for an application program, which is compiled and linked with the appropriate language-specific facilities. The SQL statements are executed in the context of the application program.
- Interactive SQL (ISQL) is a terminal-based interactive module offering a subset of the full MIMER/SQL language for direct creation and manipulation of the database.
- Batch SQL (BSQL) is a line-oriented interface designed for use from command files and scripts.
- ODBC is a database independent interface specified by Microsoft. Through ODBC MIMER can support many of the tools available in the Microsoft Windows environment.

This manual describes the usage of SQL embedded in application programs, and provides, together with the MIMER/SQL Reference Manual, the complete reference material for ESQL. The interactive and batch SQL facilities are described in the MIMER/SQL Interactive User's Manual.

2 EMBEDDING MIMER/SQL IN APPLICATION PROGRAMS

2.1 The scope of embedded MIMER/SQL

The following groups of SQL statements are common to the interactive and embedded languages:

- Data manipulation statements for reading or changing the content of the database. These are basically similar between interactive and embedded SQL, but differ in certain details as a result of the different environments in which the statements are used.
- Transaction control statements for grouping database operations in transactions (indivisible units of work).
- Security control statements for allocating privileges and access rights to users of the system. These are identical between interactive and embedded SQL.
- Data definition statements for creating and altering objects in the database. These are identical between interactive and embedded SQL.
- Access control statements for identifying the current user of the system.

The utility commands, such as LOAD, DESCRIBE and LIST, provided with MIMER Interactive SQL (ISQL) are not included in the Embedded SQL interface.

A number of statements are necessary in embedded SQL but have no counterpart in the interactive language. These are

- Declarations, required for establishing communication between the application program and the SQL statements.
- Dynamic SQL statements, which provide special facilities for formulating SQL statements during execution of an application program.
- Diagnostic statement GET DIAGNOSTICS, used to get information about the most recently executed SQL statement.

In the MIMER/SQL Reference Manual, statements are identified as valid in ESQL, ISQL or both.

2.2 General principles for embedding SQL statements

2.2.1 Host languages

Statements in MIMER/SQL may be embedded in application programs written in C, COBOL or FORTRAN-77. The machine specific User's Guide describes the ESQL preprocessors available for a specific environment. The basic principles for writing embedded SQL programs are the same in all languages. Information given in this manual applies to all languages unless otherwise explicitly stated. Language-specific information is detailed in Appendix A.

All ESQL statements are embedded in the same way in the application program.

2.2.2 Identifying SQL statements

SQL statements are included in the host language source code exactly as though they were ordinary host language statements (i.e. they follow the same rules of conditional execution, etc, which apply to the host language).

SQL statements are identified by the leading keywords 'EXEC SQL' (in all languages) and are terminated by a language-specific delimiter. Every separate SQL statement must be delimited in this way. 'Blocks' of several statements may not be written together within one set of delimiters. For instance, two consecutive DELETE statements must be written (in COBOL) as:

```
EXEC SQL DELETE FROM HOTEL      END-EXEC .
EXEC SQL DELETE FROM ROOMS     END-EXEC .
```

and not

```
EXEC SQL DELETE FROM HOTEL
        DELETE FROM ROOMS     END-EXEC .
```

Single SQL statements can however be split over several lines, following the host language rules for line continuation. The following embedded statement is thus acceptable in a FORTRAN program (the continuation mark is a '+' in column 6 on the second line):

```
EXEC SQL DELETE FROM HOTEL
+           WHERE CITY = 'SAN FRANCISCO'
```

The keywords 'EXEC SQL' may not be split over more than one line.

2.2.3 Comments

Comments may be written in the embedded SQL program according to the rules for writing comments in the host language. Thus comments may be written within an SQL statement if the host language accepts comments within host language statements. The following statement is valid in C:

```
exec sql SELECT HOTELNAME /* 20 characters */  
          FROM   HOTEL;
```

The keywords 'EXEC' and 'SQL' may not be separated by a comment.

2.2.4 Recommendations

The following recommendations are imposed by the use of embedded SQL:

- Variable names beginning with the letters 'SQL' should be avoided (except for SQLSTATE and SQLCODE, which should be used when appropriate).
- Subroutine or subprogram names ending with a number should be avoided.
- Language-specific restrictions are described in Appendix A.

2.3 Processing embedded SQL

2.3.1 Preprocessing

An application program using embedded SQL statements must first be preprocessed before it can be passed through the host language compiler, since the host language itself does not recognize the EXEC SQL identifier. Preprocessors are available for each of the host languages supported.

The preprocessor is essentially a translator which parses embedded SQL statements and converts them to the appropriate source code calls which pass the SQL statements to the MIMER database manager. The SQL statements in the original program code are retained as comments. The output from the preprocessor is human-readable source code retaining a large part of the structure and layout of the original program.

Note: The application programmer should never attempt to modify the output from the preprocessor directly. Any changes which may be required in a program should be introduced into the original embedded SQL source code. Sysdeco Mimer cannot accept any responsibility for the consequences of modifications to the preprocessed code.

The preprocessor checks the syntax and to some extent the semantics of the embedded SQL statements (see Chapter 8 for a more detailed discussion of how errors are handled). Syntactically invalid statements cannot be preprocessed and the source code must be corrected.

Options available for the preprocessors are described in Appendix B.

2.3.2 Processing embedded SQL - the compiler

The output from the SQL preprocessor is compiled in the usual way using the appropriate host language compiler, and linked with the appropriate routine libraries. At run-time, database management requests are passed to the SQL compiler responsible for implementing the SQL functions in the application program.

The SQL compiler performs two functions:

- SQL statements are checked semantically against the data dictionary.
- Operations performed against the database are optimized (i.e. internal routines determine the most efficient way to execute the SQL request, with regard to the existence of secondary indexes and the number of rows in the tables addressed by the statement). The programmer does not need to worry, for instance, about the order in which tables are addressed in a complex selection condition. This optimization process is completely transparent.

A statement which occurs once in the program source code but is repeatedly executed in loop constructions (of any kind) is compiled once only by the SQL compiler. However, an SQL statement which is explicitly repeated in the source code is compiled separately each time. It is therefore advantageous to use loop constructions wherever possible for repeated SQL statements.

Note that since all SQL statements are compiled at run-time, there can be no conflict between the state of the database at the times of compilation and execution. Moreover, the execution of SQL statements is always optimized with reference to the current state of the database.

2.4 Essential program structure

All application programs using embedded MIMER/SQL must include certain basic components, summarized below in the order in which they appear in a program. (Declarations must be written first in the program; the other components are named in logical execution order).

- Host variable declarations. Any host variables used in SQL statements must be declared inside a so-called SQL DECLARE SECTION. This is described in more detail in Chapter 3.
- The status information variable SQLSTATE must be declared inside the SQL DECLARE SECTION. This variable provides the application with status information for the most recently executed SQL statement.
- Executable SQL statements. This is the 'body' of the program, and performs the required operations on the database. Normally, these begin with connecting to MIMER and end with statements to close all cursors, end the current transaction, and disconnect from MIMER.

The following table summarizes the functions for data manipulation in interactive and embedded SQL.

Operation	Interactive	Embedded
Retrieve data	SELECT generates a result table directly	SELECT is used to declare a cursor. The cursor must be opened and positioned. Data is retrieved into host variables one row at a time with FETCH. Alternative: SELECT INTO retrieves a single-row result set directly into host variables
Update data	UPDATE operates on a set of rows/columns	UPDATE operates on a set of rows or columns UPDATE CURRENT operates on a single row through a cursor
Insert data	INSERT inserts one or many rows at a time	INSERT inserts one or many rows at a time
Delete data	DELETE operates on a set of rows	DELETE operates on a set of rows DELETE CURRENT operates on a single row through a cursor

Many SQL statements (e.g. data definition statements) are simply embedded in their logical place in the application program and are executed without direct reference to other parts of the program. Some features of ESQL however require special consideration, and are dealt with in detail in the chapters that follow:

- Access authorization through the use of user and program ids.
- Data manipulation statements which require the use of cursors (FETCH, UPDATE CURRENT, DELETE CURRENT). These together with cursor handling statements are probably the most commonly used statements in ESQL.
- Transaction control, which is essential for a consistent database.
- Dynamic SQL, which is a special set of statements allowing an application program to process SQL statements entered by the user at run-time.
- Exception handling, which controls the action taken when, for instance, the end of a result set is reached.

3 COMMUNICATING WITH THE APPLICATION PROGRAM

Information is transferred between the embedded SQL application program and the MIMER database manager in four ways:

- through host variables used in SQL statements.
- through the status variable SQLSTATE.
- through the diagnostics area, accessed by the SQL statement GET DIAGNOSTICS.
- through an SQL descriptor area (dynamic area).

3.1 Host variable usage

Host variables are used in SQL statements to pass values between the database and the application program.

3.1.1 Declaring host variables

All variables used in SQL statements must be declared for the preprocessor, by enclosing the variable declarations between the SQL statements BEGIN DECLARE SECTION and END DECLARE SECTION. Any variables declared outside the SQL DECLARE SECTION will not be recognized by the preprocessor. Variables are declared within the section using the normal host language syntax; for instance the following example in C declares only the character variables **user** and **passwd** for use in SQL statements:

```
int rc,
    pf,
    count;
exec sql BEGIN DECLARE SECTION;
char user[19],
    passwd[19];
exec sql END DECLARE SECTION;
```

Variables which are not used in SQL statements may also be declared in the SQL DECLARE SECTION. This will however extend the symbol table established by the preprocessor more than is necessary.

The use of array variables is currently not supported in MIMER/SQL statements.

3.1.2 Using variables in statements

Host variables may be used to:

- receive information from the database (SELECT INTO and FETCH statements)
- to assign values to columns in the database (UPDATE and INSERT statements)
- to manipulate information taken from the database or contained in other variables (in expressions)
- to keep cursors and statement identifiers for dynamic SQL
- to provide values for comparative evaluations (predicates).

In all these contexts, the data type of the host variable or database column must be compatible with the data type of the corresponding database value or host variable. General considerations of data type compatibility may be found in the MIMER/SQL Reference Manual, 'Basic SQL syntax rules'. Host language specific aspects are described in Appendix A of this manual.

If you have an INTEGER column containing values that does not fit into the largest integer variable allowed on your machine (remember that MIMER supports INTEGER values with a precision of up to 45 digits), you must use a character string host variable for that column. In this case, MIMER automatically performs the necessary conversions.

Host variable names are preceded by a colon when used in SQL statements (see the MIMER/SQL Reference Manual). Note that the colon is not part of the host variable name, and should not be used when the variable is referenced in host language statements. For example:

```
exec sql  SELECT column
          INTO    :var
          FROM    table
          WHERE   condition;
IF var < limit THEN ...
```

3.1.3 Indicator variables

ESQL uses *indicator variables* associated with main variables to handle NULL values in database tables. Indicator variables should be declared as 'short' variables in the SQL DECLARE SECTION. See Appendix A, under 'Host Variable' - 'Declaration', for a description of how indicator variables should be declared in the specific host languages.

Indicator variables are used in SQL statements by concatenating the name of the indicator variable preceded by a colon to the main variable name, or by using the keyword INDICATOR

```
          :main_variable:indicator_variable
or
          :main_variable INDICATOR :indicator_variable
```

Transfer from tables to host variables

When a NULL value is retrieved into a host variable by a FETCH or SELECT INTO statement, the value of the main variable remains unchanged and the value of the indicator variable is set to -1. An error occurs if the main variable is not associated with an indicator variable in the SQL statement. It is therefore recommended as a precaution that indicator variables are used for all columns which are not defined as NOT NULL in the database. It is advisable that the application makes an explicit test for -1, and not just checks that it is a negative value in the indicator variable, since negative values other than -1 may come to indicate another meaning in future releases of MIMER/SQL.

When a non-null value is assigned to a main variable associated with an indicator variable, the indicator variable is set to zero or a positive value. A positive value indicates that the value assigned to a main character variable was truncated, and gives the length of the original value before truncation.

Transfer from host variables to tables

When the host variable associated with an indicator variable is used to assign a value to a column, the value assigned is NULL if the value of the indicator variable is set to -1. In such a case the value of the main variable is irrelevant. If the indicator variable has a value of zero or a positive value, or if the main variable is not associated with an indicator variable, the value of the main variable itself is assigned to the column.

3.2 The SQLSTATE variable

The SQLSTATE variable provides the application, in a standardized way, with return code information about the most recently executed SQL statement. SQLSTATE must be declared between the BEGIN DECLARE SECTION and the END DECLARE SECTION (i.e. in the SQL declare section), as a 5 character long string (excluding any terminating null byte). The return codes provided by SQLSTATE can contain digits and capital letters.

SQLSTATE consists of two fields. The first two characters of SQLSTATE indicates a class, and the following three characters indicates a subclass. Class codes are unique, but subclass codes are not. The meaning of a subclass code depends on the associated class code. To determine the category of the result of a SQL statement, the application can test the class of SQLSTATE according to the following:

<u>SQLSTATE class</u>	<u>Result category</u>
'00'	Success
'01'	Success with warning
'02'	No data
Other	Error

For a list of SQLSTATE values see Appendix C.

3.3 The diagnostics area

The diagnostics area holds status information for the most recently executed SQL statement. There is always one diagnostics area for an application, no matter how many connections the application holds. Information from the diagnostics area is selected and retrieved by the GET DIAGNOSTICS statement. The syntax for GET DIAGNOSTICS is described in Chapter 8. The GET DIAGNOSTICS statement does not change the contents of the diagnostics area, although it does set SQLSTATE.

The diagnostics area contains either supplementary status information for the latest executed SQL statement, or exception information if the SQL statement resulted in an error.

For a description of the diagnostics area, see Section 8.3.

The message text retrieved from the diagnostics area will include an internal MIMER return code, as well as a descriptive message. See Appendix C for a list of internal MIMER return codes.

3.4 The SQL Descriptor Area

An SQL descriptor area is used to hold descriptive information required for execution of dynamic SQL statements. SQL descriptor areas are allocated and maintained by embedded SQL statements, described in the MIMER/SQL Reference Manual.

The SQL descriptor area is discussed in detail in Chapter 7.

4 IDENTS AND DATABASE CONNECTIONS

An ident in a MIMER system is an authorized user of the system, or the collective identity of a group of users sharing common privileges. Idents connect to a database through the `CONNECT` statement (or the `ENTER` statement in the case of program idents, see below).

A *database* in MIMER version 7 refers to the complete collection of databanks which may be accessed from one MIMER system. A new feature in version 7 is the ability to change between different connections (i.e. access different databases) from within the same application program. A program may have several database connections open simultaneously, although only one is active at any one time.

4.1 MIMER idents

There are four kinds of ident in MIMER:

USER idents are authorized to log on to any MIMER module, by using the `CONNECT` statement in an application program or by entering the correct ident name and password in an interactive environment. Any privileges a user ident holds may be exercised once the ident has logged on. User idents are generally associated with specific physical individuals authorized to use the system.

OS_USER is an ident type which allows the user currently logged in to the operating system to log on to the MIMER database without providing a username or password. If an `OS_USER` ident is defined with a password in MIMER, the ident may connect to MIMER in the same way as any other user ident (i.e. by providing a username and password). An `OS_USER` ident is subject to the same access restrictions as any other user ident.

- PROGRAM** idents may not log on to MIMER, but may be entered from within an application program or interactive environment by using the ENTER statement. The ENTER statement should be preceded by a successful CONNECT statement. If no successful CONNECT has been performed, an implicit connection will be attempted before the ENTER the program is entered. If the implicit connection fails, the ENTER statement will also fail. See Section 4.3.1 for a description of implicit connections. Entering a program ident is analogous to logging on as a user ident, in that the program ident gains access to the system and any privileges the ident holds become applicable. Program idents are generally associated with specific functions within the system, not with physical individuals.
- GROUP** idents are collective identities for groups of user or program idents. Any privileges granted to or revoked from a group ident automatically apply to all members of the group. Any ident can be a member of as many groups as required, and one group can include any number of members. Group idents provide a facility for organizing the privilege structure in the database system.

USER, OS_USER and PROGRAM idents are authorized users of the system. Every USER and PROGRAM ident has a unique ident name and a private password which must be correctly supplied to the CONNECT or ENTER statement in application programs. An OS_USER may access the database without explicitly providing a username or password on condition that the username for the user currently logged in to the operating system correspond to the definition of an OS_USER in the MIMER database.

4.2 Access to the database

The access of each ident to the database is defined by privileges granted within the system. The privileges are grouped as follows:

System privileges

DATABANK gives the right to create databanks
 IDENT gives the right to create idents

Object privileges

TABLE gives the right to create tables in a specified databank
 EXECUTE gives the right to enter (connect to) a specified program ident
 MEMBER grants membership in a specified group ident

Access privileges

SELECT gives the right to read the table contents
 INSERT gives the right to add new rows to the table
 DELETE gives the right to remove rows from the table
 UPDATE gives the right to update the contents of the table
 REFERENCES gives the right to use the primary key or alternate keys of the table as a foreign key from another table

System privileges are automatically granted to the system administrator at installation, and may be passed on to other idents. Object and access privileges are initially granted only to the creator of an object. The creator may however grant the privileges on to other idents.

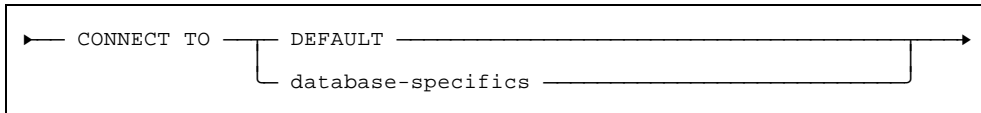
All privileges may be granted with or without GRANT OPTION, which controls the right of the receiving ident to grant the privilege on to another ident.

Certain operations are not controlled by explicit privileges, but may only be performed by the creator of the object involved. These operations include ALTER (with the exception of ALTER IDENT, which may be performed by either the ident himself or by the creator of the ident), DROP, and COMMENT. Similarly, privileges may only be revoked by their grantor.

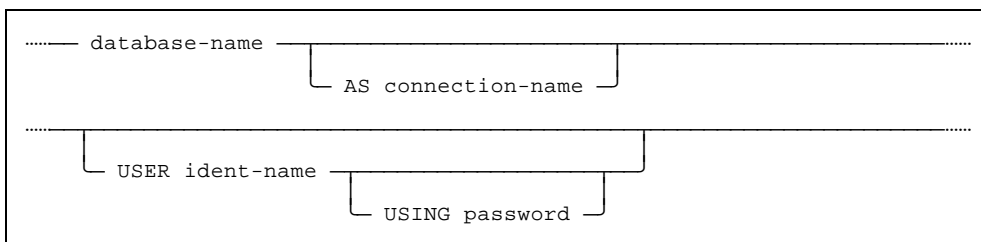
4.3 Connecting to a database

4.3.1 Connecting

Only idents of type USER are allowed to log on to MIMER. Logging on is requested from an application program with the CONNECT statement, with the syntax



where *database-specifics* is



The *database-name*, *connection-name*, *ident-name* and the *password* can be entered either as a host variable or as a literal.

The CONNECT statement establishes a connection between a user and a database. The user may be a USER ident (in which case the password must be provided) or an OS_USER (if such an ident has been created in the MIMER database). To connect as an OS_USER, provide an empty ident name string.

The database may be specified as a database name or as the keyword DEFAULT. System databank locations corresponding to named databases and to the DEFAULT database are listed in the SQLHOSTS file in the current environment (see MIMER System Management Handbook and machine-specific User's Guide for more details). If the database name is given as an empty string, the DEFAULT database is used.

The database may be given an explicit connection name for use in DISCONNECT and SET CONNECTION statements. If no explicit name is given, the database name is used as connection name.

Normally, CONNECT should be the first executable SQL statement in an application program using ESQL. However, if an executable statement is issued before any connection has been established in the current application, an implicit connection attempt will be made to the DEFAULT database using the OS_USER ident. The attempt will be successful if the DEFAULT database is valid and the current operating system user is also defined as an OS_USER ident in MIMER. If an implicit connection has previously been established in the application, but there is no current connection, issuing an executable statement will result in a new attempt to make an implicit connection. However, if an explicit connection has previously been established in the application, but there is no current connection, issuing an executable statement will cause an error.

4.3.2 Changing connection

A connection established by a successful CONNECT statement is automatically active. An application program may make multiple connections to the same or different databases using the same or different idents, provided that each connection is identified by a unique connection name. Only the most recent connection is active. Other connections are dormant, and may be made active by the SET CONNECTION statement. Resources such as cursors used by a connection are saved when the connection is inactivated, and are restored by the appropriate SET CONNECTION statement.

The statement sequence below connects to a user-specific database as a specified ident name and to the DEFAULT database as OS_USER. The user-specific connection is initially active. Then the DEFAULT connection is activated. Finally the user-specific connection is activated again using SET CONNECTION.

```
exec sql CONNECT TO 'db' AS 'common' USER 'ident' USING 'pswd';
...
exec sql CONNECT TO DEFAULT;
...
-- Set activate connection to COMMON
exec sql SET CONNECTION 'common';
```

Note that if different connections are made with different idents, the apparent access rights of the application program may change when the current connection is changed.

4.3.3 Disconnecting

The DISCONNECT statement breaks the connection between a user and a database and frees all resources allocated to that user for the specified connection (all cursors are closed and all compiled statements are dropped). The connection to be broken is specified as the connection name or as one of the keywords ALL, CURRENT or DEFAULT. No transaction may be active when DISCONNECT is performed.

A connection does not have to be active in order to be disconnected. If an inactive connection is broken, the application still has uninterrupted access to the database through the current (active) connection, but the broken connection is no longer available for activation with SET CONNECTION.

If the active connection is broken, the application program cannot access the database until a new `CONNECT` or `SET CONNECTION` statement is issued. Note the distinction between breaking a connection with `DISCONNECT` and making a connection inactive by issuing a `CONNECT` or `SET CONNECTION` for a different connection. A broken connection has no saved resources and cannot be reactivated by `SET CONNECTION`.

The table below summarizes the effect on the connection 'con1' of `CONNECT`, `DISCONNECT` and `SET CONNECTION` statements depending on the state of the connection

Statement	con1 non-existent	con1 current	con1 dormant
<code>CONNECT TO db1 AS con1</code>	con1 current	error- connection already exists	error- connection already exists
<code>DISCONNECT con1</code>	error- connection doesn't exist	con1 disconnected	con1 disconnected
<code>SET CONNECTION con1</code>	error- connection doesn't exist	ignored	con1 current
<code>CONNECT TO db2 AS con2</code>	-	con1 made dormant	con1 unaffected
<code>DISCONNECT con2</code>	-	con1 unaffected	con1 unaffected
<code>SET CONNECTION con2</code>	-	con1 made dormant	con1 unaffected

4.3.4 Program idents – ENTER and LEAVE

Program idents may be entered from within an application program by using the `ENTER` statement. This statement should be issued in a context where a user is already connected: program idents cannot connect directly to the system. The syntax of the `ENTER` statement is:

```
← ENTER ident-name IDENTIFIED BY password →
```

Both the *ident-name* and the *password* can be entered either as a host variable or as a literal.

When a program ident is entered, any privileges granted to that ident become current and privileges belonging to the previous ident (i.e. the ident issuing the `ENTER` statement) are suspended. However, any cursors opened by the previous ident remain open.

Program idents are disconnected with the `LEAVE` statement. If `LEAVE` is requested with the optional keyword `RETAIN`, the full environment of the program ident being left is kept. Cursors left open by the program ident are deactivated but not closed, and retain their positions in the respective result tables. The environment is restored if the program ident is re-entered.

If `LEAVE` is requested without `RETAIN`, the environment of the program ident being left is dropped. This means that all cursors and compiled statements are destroyed.

Note the distinction between leaving a program ident with the option `RETAIN` and entering a new program ident. Both operations save the environment of the program ident, but cursors left open at `ENTER` may still be used, while those left open at `LEAVE RETAIN` are inaccessible until the program ident is re-entered.

The statements `ENTER` and `LEAVE` may not be issued within transactions (see Chapter 6).

5 ACCESSING DATABASE TABLES

5.1 Retrieving data

Data is retrieved from database tables with the `FETCH` statement, which fetches the values from an individual row in a result set into host variables. The result set is defined by a `SELECT` construction used in a cursor declaration; the cursor may be thought of as a pointer which moves through the rows of the result set as successive `FETCH` statements are issued. Variables in predefined communication areas are used to report the outcome of the `FETCH` (e.g. to indicate that the end of the result set has been reached). Data retrieval thus involves several steps in the application program code:

- declaration of host variables to hold data and communication areas
- declaration of a cursor with the appropriate `SELECT` conditions
- opening the cursor
- performing the `FETCH`
- closing the cursor

These steps are built into the application program as shown in the general framework below (only SQL statements are shown in the framework):

```
exec sql BEGIN DECLARE SECTION;
...
exec sql END DECLARE SECTION;

exec sql DECLARE cursor-name CURSOR FOR select-statement;

exec sql OPEN cursor-name;

loop as required
  exec sql FETCH cursor-name INTO :var1,:var2,...,:varn;
end loop

exec sql CLOSE cursor-name;
```

5.1.1 Declaring host variables

All host variables used to hold data fetched from the database and used in selection conditions must be declared within an `SQL DECLARE SECTION` (see Chapter 3). Indicator variables for columns which may contain `NULL` values must also be declared. The same indicator variable may be associated with different main variables at different times, but declaration of a dedicated indicator variable for each main variable is recommended for clarity.

5.1.2 Declaring the cursor

A cursor operates as a row pointer associated with a result table. A cursor is defined by the `DECLARE CURSOR` statement, and the set of rows addressed by the cursor is defined by the `SELECT` statement in the cursor declaration. Cursors are local to the program where they were declared. A cursor is given an identifying name when it is declared.

`DECLARE CURSOR` is a declarative statement, which does not result in an implicit `CONNECT` to the default database (see Chapter 4). Preprocessing the statement generates a series of parameters used by the SQL compiler but does not generate any executable code; the `SELECT` statement in the cursor declaration is executed when the cursor is opened.

It is advisable always to use an explicit list of items in the `SELECT` statement of the cursor declaration. The shorthand notations `'SELECT *'` and `'SELECT table.*'` are useful in interactive SQL, but can cause conflicts in the variable lists of `FETCH` statements if the table definition is changed.

The cursor declaration can use host variables in the `WHERE` or `HAVING` clause of the `SELECT` statement. The result table addressed by the cursor is then determined by the values of these host variables at the time when the cursor is opened. The same cursor declaration can thus address different result tables depending on when the cursor is opened, for example

```
exec sql DECLARE C1 CURSOR..;      -- cursor with host variables
set variables
exec sql OPEN C1;                  -- open one result set
...
exec sql CLOSE C1;
change variables
exec sql OPEN C1;                  -- open different result set
```

Scrollable cursors can be declared using the `SCROLL` keyword. When a cursor is declared as scrollable, records can be fetched using an orientation specification. This makes it possible to scroll through the result set with the cursor.

Cursors which are to be used only for retrieving data may be declared with a `FOR READ ONLY` clause in the `SELECT` statement. This can improve performance slightly in comparison with cursors which permit update and delete operations.

5.1.3 Opening the cursor

A declared cursor must be opened with the `OPEN` statement before data can be retrieved from the database. The `OPEN` statement evaluates the `SELECT` clause in the cursor declaration in terms of

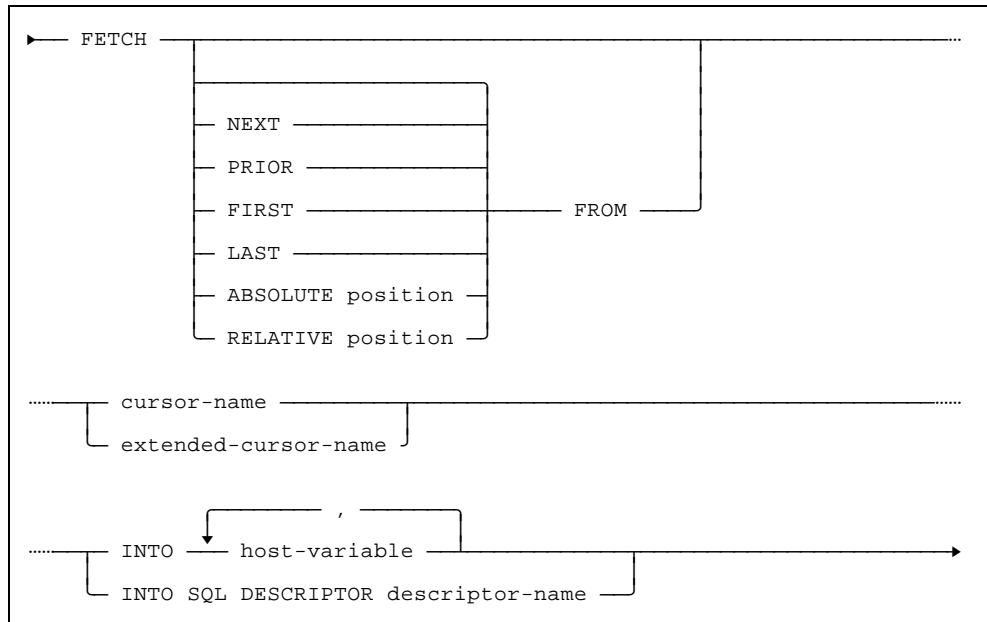
- the access rights of the current user
- the values of any host variables used in the `SELECT` clause

When the `OPEN` statement has been executed, the cursor is positioned before the first row in the result table.

`SELECT` access on tables and views addressed by the cursor is required to open a cursor.

5.1.4 Retrieving data

Once a cursor has been opened, data may be retrieved from the result table with `FETCH` statements. `FETCH` has the following syntax:



Host variables in the variable list correspond in ordering to the columns named in the `SELECT` clause of the cursor declaration. The number of variables in the `FETCH` statement may not be more than the number of columns selected. The number of variables is allowed to be less than the number of columns selected, but a 'success with warning'-code is then returned in `SQLSTATE`.

A suitably declared record structure may be used in place of a variable list in host languages where this is supported (see Appendix A).

The orientation specification is optional, and it may only be specified on a cursor declared as scrollable (except for `NEXT` which is always allowed). With the orientation specification it is possible to specify which record to fetch. The orientation specification can be one of `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE position` and `RELATIVE position`.

See MIMER/SQL Reference Manual, 'SQL statement description' for a further description of orientation specification. If orientation specified is not given, or if the cursor is not defined as scrollable, `FETCH NEXT` is implicit.

Each `FETCH` statement moves the cursor to the specified row in the result table before retrieving column values. In strict relational algebra, the ordering of tuples in a relation (the formal equivalent of rows in a table) is undefined. The `SELECT` statement in the cursor declaration may include an `ORDER BY` clause if the ordering of rows in the result table is important to the application (note however that a cursor declared with an `ORDER BY` clause cannot be used for updating table contents). If no `ORDER BY` clause is specified, the ordering of rows in the result table is unpredictable.

Note that the variables into which data is fetched are specified in the `FETCH` statement, not in the cursor declaration. In other words, data from different rows in the result table may be fetched into different variables.

When there are no more rows to fetch, the exception condition NOT FOUND will be fulfilled. The following construction thus fetches rows successively until the result table is exhausted:

```
exec sql DECLARE C1 CURSOR FOR select-statement;
exec sql OPEN C1;

exec sql WHENEVER NOT FOUND GOTO done;
loop
    exec sql FETCH C1 INTO :var1,:var2,...,:varn;
end loop

done:
exec sql CLOSE C1;
```

The access rights for a user are checked when the cursor is opened, and they remain unchanged for that cursor until the cursor is closed. For example, if an application program declares and opens a cursor, and then SELECT access on the table is revoked from the user running the program, data can still be fetched from the result table as long as the cursor remains open. Any subsequent attempt to open the same cursor will however fail.

5.1.5 Closing the cursor

An opened cursor remains open until it is closed with the one of the CLOSE, COMMIT or ROLLBACK statements, or until the current connection is disconnected. Once a cursor is closed, the result table is no longer accessible. The cursor declaration remains valid, however, and a new cursor may be opened with the same declaration (note that the result table addressed by the new cursor may not be the same if the contents of the database or the values of variables used in the declaration have changed).

Normally, resources used by the cursor remain allocated when the cursor is closed, and will be used again if the cursor is re-opened. The optional form CLOSE cursor-name RELEASE deallocates cursor resources. Use of CLOSE RELEASE is recommended in application programs which open a large number of cursors, particularly where system resources are limited. Note that the use of CLOSE RELEASE slows down performance, since it requires that new resources are allocated at the next OPEN. For this reason it should not be used unless it is really needed.

Cursors are local to a connection and remain open but dormant when the connection is made dormant. The state of dormant cursors is fully restored (including result set addressed and position in the result set) when the connection is reactivated. Cursors are however closed and cursor resources are deallocated when a connection is disconnected. Note however that cursors opened in a program ident context are closed and resources deallocated when LEAVE is executed within the same connection, unless LEAVE RETAIN is specified.

5.2 Retrieving single rows

If the result of a `SELECT` statement is known to be a single row, the `SELECT INTO` statement may be used as an alternative to fetching data through a cursor. This is a much simpler programming construction, since cursors are not used and the only requirement is that host variables used in the `SELECT INTO` statement are declared in the `DECLARE SECTION`. However, there are two disadvantages associated with `SELECT INTO`:

- An error occurs if the result table addressed by the search condition contains more than one row. In other words, `SELECT INTO` can only be reliably used when there is no possibility of a multi-row result table (essentially when the search condition includes a `UNIQUE` or `PRIMARY KEY` column).
- Execution of the `SELECT INTO` statement involves a check that the result table contains a single row, which may incur unnecessary overhead. Even if it is known that the result row is unique, a single `FETCH` operation through a cursor may be a more efficient implementation.

Use of a `SELECT INTO` statement is justified when the result table may contain several rows, but it is a condition for continued execution of the application program that the result row is unique. With a cursor, this would require a construction which checked that one and only one `FETCH` operation could be performed (alternatively, use a separate `SELECT COUNT` with the same search condition as the cursor). In such a case, a `SELECT INTO` statement with a check on the return code (see Chapter 8) is probably the preferred solution.

5.3 Retrieving data from multiple tables

Data can be retrieved from multiple tables in embedded SQL by addressing several tables in the `SELECT` statement of the cursor declaration, in the same way as in interactive SQL. The preprocessor generates a `SELECT` statement addressing multiple tables, which is optimized by the SQL optimizer when the cursor is opened. For example:

```
exec sql DECLARE C1 CURSOR FOR SELECT ... FROM A, B
WHERE A.X=B.Y;
exec sql OPEN C1;
...
```

An alternative way to link information between tables could be to define the search condition for one cursor in terms of a variable fetched through the other cursor:

```
exec sql DECLARE C1 CURSOR FOR SELECT X FROM A;
exec sql DECLARE C2 CURSOR FOR SELECT ... FROM B WHERE Y=:hostx;

exec sql OPEN C1;
exec sql FETCH C1 INTO :hostx;
exec sql CLOSE C1;

exec sql OPEN C2;
exec sql FETCH C2;
...
```

When considering the two alternatives, the first one is preferred. The reason for this is:

- The SQL optimizer gets the full information about the query that it is supposed to return a result set for. In this way the optimizer can make more use of statistical information, and it can thereby optimize the query to execute in a more efficient way.
- The application will require less resources in form of open cursors.
- The second alternative will cause more communication over the internal multi interface in MIMER.
- If the application is run in a client/server environment, the second alternative will cause more communication over the network, since it will send data over the net which is only used to determine which data from the second cursor that will be selected, and is of no actual interest to the application.
- The application will be more compact and easier to understand and maintain.

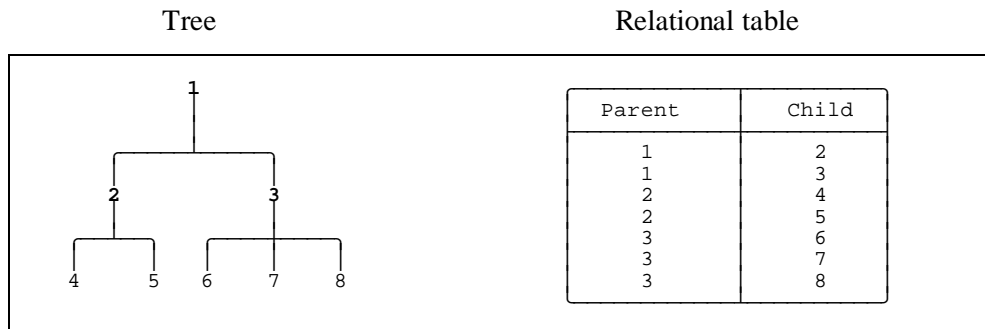
5.3.1 The 'Parts explosion' problem

A special case of data retrieval from multiple tables is the use of stacked cursors to fetch data from logical copies of the same table, in which way the so called "Parts explosion" problem can be solved. The same cursor may be opened several times in succession in the same application program: each previous instance of the cursor is saved on a stack and is restored when the following instance is closed. A FETCH statement refers to the most recently opened instance of a cursor. Each instance of the cursor addresses an independent result table, and the position of each cursor in its own result table is saved on the stack. Note that the result tables addressed by different instances of the cursor may differ according to the conditions prevailing when the cursor instance was opened.

The state of the cursor stack need to be controlled by the application. A counter can be used to indicate if there are more instances of the cursor remaining on the stack. See the example on the next page.

Stacked cursors are typically used in application programs which traverse a tree structure stored in the database. For example (this is a simplified variant of the 'parts explosion' problem):

Example: Traverse a tree structure and print out the leaf nodes.



```

procedure TRAVERSE;

    integer CSTACK, LASTC;
EXEC SQL BEGIN DECLARE SECTION;
    integer PARENT, CHILD;
EXEC SQL END DECLARE SECTION;
begin
    EXEC SQL DECLARE CTREE REOPENABLE CURSOR FOR
        SELECT PARENT, CHILD
        FROM TREE
        WHERE PARENT = :PARENT;
    CSTACK := 1;
    LASTC := 1;
    PARENT := 1;                                -- start at root node
    EXEC SQL OPEN CTREE;
loop
    EXEC SQL FETCH CTREE INTO :PARENT, :CHILD;
    if SQLSTATE = "02000" then                  -- no more children
        EXEC SQL CLOSE CTREE;                  -- pop the parent
        CSTACK := CSTACK-1;
        exit when CSTACK = 0;
        if CSTACK >= LASTC then
            print(PARENT);                      -- write leaf node
        end if;
        LASTC := CSTACK;
    else                                        -- step to next level
        PARENT := CHILD;
        EXEC SQL OPEN CTREE;                  -- stack the current parent
                                                -- and open new level
        CSTACK := CSTACK+1;
    end if;
end loop;
end TRAVERSE;

```

The counters CSTACK and LASTC keep track of the number of stacked cursor levels and the latest level in the tree hierarchy respectively.

5.4 Entering data into tables

5.4.1 Cursor-independent operations

The SQL statements INSERT, UPDATE and DELETE embedded in application programs operate on a set of rows in a table or view in exactly the same way as in interactive SQL. Host variables may be used in the statements to supply values or set search conditions, for example

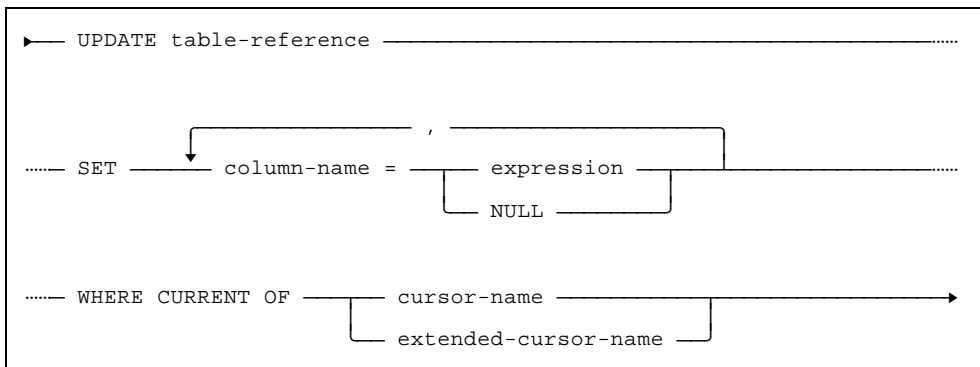
```
exec sql UPDATE BOOKADM.FREEROOMS
      SET   FREECOUNT = FREECOUNT-1
      WHERE HOTELCODE = :HOTCOD
      AND   ROOMTYPE  = :BOOK_ROOMT
      AND   ON_DATE   >= :BOOK_ARRIVE
      AND   ON_DATE   < :BOOK_DEPART;
```

From the standpoint of the application program, each statement is a single indivisible operation, regardless of how many columns and rows are affected.

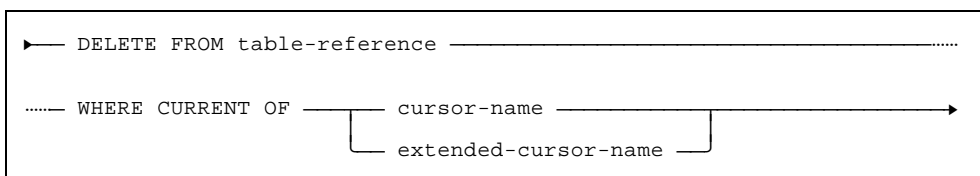
5.4.2 Updating and deleting through cursors

The UPDATE CURRENT and DELETE CURRENT statements allow update and delete operations respectively to be controlled on a row-by-row basis from an application program. These statements operate through cursors, which are declared and opened as described above for FETCH.

The syntax of the UPDATE CURRENT and DELETE CURRENT statements is:



and



These statements operate on the current row of the cursor referenced in the statement. If there is no current row (e.g. the cursor has been opened but not yet positioned with a FETCH statement), an error is returned.

UPDATE CURRENT changes the content of the current row according to the SET clause in the statement, but does not change the position of the cursor. Two consecutive UPDATE CURRENT statements will therefore update the same row twice.

DELETE CURRENT deletes the current row and does not move the cursor; after a DELETE CURRENT statement, the cursor is positioned "between rows" and there is no current row. The cursor must be moved to the next row with a FETCH statement before any other operation can be performed through the cursor.

For both UPDATE CURRENT and DELETE CURRENT statements, the table name as used in the statement must be exactly the same as the table name addressed in the cursor declaration. The cursor must also address an updatable result table, and for UPDATE CURRENT must include a FOR UPDATE OF clause (which specifies which columns may be updated). Cursors are *not* updatable if the SELECT statement in the cursor declaration contains any of the following features at the top level (i.e. not in a subselect) of the statement:

- reference to more than one table in the FROM clause
- reference to a read-only view in the FROM clause
- the keyword DISTINCT
- set-functions in the SELECT list (AVG, COUNT, MAX, MIN, SUM)
- arithmetic or string concatenation expressions in the SELECT list
- a GROUP BY clause
- an ORDER BY clause
- the UNION keyword
- the result-set of an explicit inner or outer JOIN

When to use UPDATE CURRENT, DELETE CURRENT

UPDATE CURRENT and DELETE CURRENT statements are useful for manipulating single rows in interactive applications where rows are displayed, and the user decides which rows to delete or update. The example below illustrates the program framework for such an operation (the construction is similar for a DELETE CURRENT operation):

```

exec sql BEGIN DECLARE SECTION;
...
exec sql END DECLARE SECTION;
...
exec sql DECLARE C1 CURSOR FOR ... .. FOR UPDATE OF ...;
...
exec sql OPEN C1;
exec sql WHENEVER NOT FOUND GOTO done;
loop
  exec sql FETCH C1 INTO :var1,:var2,...,:varn;
  display var1,var2,...,varn;
  prompt "Update this row?";
  if answer = "yes" then
    prompt "Give new values";
    exec sql UPDATE tab SET col1 = :newval1, col2 = newval2, ...
      WHERE CURRENT OF C1;
    display "row updated";
  end if;
  prompt "Display next row?";
  if answer = "no" then
    exit;
  end if;
end loop;

done:
exec sql CLOSE C1;

```

Cursor-independent UPDATE or DELETE statements should be used wherever possible when the set of rows to be updated or deleted can be defined completely in terms of a WHERE clause.

6 TRANSACTION HANDLING AND DATABASE SECURITY

6.1 Transaction principles

A transaction is an 'atomic' operation which may not be divided into smaller operations. Two transaction phases exist: *build-up*, during which the database operations are requested, and *commitment*, during which the operations are actually executed against the database. Transaction build-up may be started explicitly or implicitly (see Section 6.2.1); commitment is always requested explicitly with the COMMIT statement. In interactive application programs, build-up takes place typically over a time period determined by the user, while commitment is an internal process on a time-scale determined by machine operations.

6.1.1 Concurrency control

Since MIMER uses optimistic concurrency control, deadlocks never occur (see Section 6.1.2 for a further discussion of deadlocks). How optimistic concurrency control works in MIMER is described below.

During build-up, changes requested in the contents of the database are kept in a *write-set* and are not visible to other users of the system, and the database remains fully accessible to all users. The application program in which build-up occurs may see the database as though the changes were already made (by read-through-write-set, see below). Changes requested during transaction build-up become visible to other users when the transaction is successfully committed.

During build-up, a *read-set* records the state of the database as seen at the time of each operation (including intended changes). If the state of the database at commitment is inconsistent with the read-set, a conflict is reported and the transaction is rolled back (i.e. the write-set is erased and no changes are made to the database). This can happen if for instance a transaction asks to update a row which is deleted by another user after build-up is requested but before the transaction is committed. The application program is responsible for taking appropriate action if a transaction conflict occurs.

Because of the nature of this concurrency control protocol, it is important that some of the implications are understood.

For example, a transaction can be built up over an extended period of time, either because of a large number of interactions with the user during the build up, or because the user leaves the terminal part way through the build up of a transaction.

A transaction that exists for a long elapsed time has a greater chance of conflicting with changes made by other users, than a transaction that has a short elapsed time.

At the other extreme, an application that immediately commits every executed SQL statement will seldom meet any conflicts, but will incur unnecessary overhead.

In general:

- keep transactions as short as is reasonably possible
- if possible, keep interactive user dialogs outside of transactions

A common situation that can generate unnecessarily large read-sets is the following: An application program reads through the rows in a table in a loop structure, with a conditional exit to update a row on user intervention. It is tempting to simply place a COMMIT after the update statement:

```
loop
  exec sql FETCH C1 INTO :var1,:var2,...,:varn;
  display var1,var2,...,varn;
  prompt "Update row?";
  exit when answer = "yes";
end loop

exec sql UPDATE table SET ... WHERE CURRENT OF C1;
exec sql COMMIT;
```

The FETCH loop can create a large read-set while waiting for the user update request, risking transaction conflict at the UPDATE. A tempting solution for this problem might be:

```
loop
  exec sql FETCH C1 INTO :var1,:var2,...,:varn;
  display var1,var2,...,varn;
  prompt "Update row?";
  exit when answer = "yes";
  exec sql ROLLBACK;
end loop;

exec sql UPDATE table SET ... WHERE CURRENT OF C1;
exec sql COMMIT;
```

But since ROLLBACK closes all cursors, this will not work. Instead something like the following is a better solution:

```
exec sql SET TRANSACTION START EXPLICIT

loop
  exec sql FETCH C1 INTO :var1,:var2,...,:varn;
  display var1,var2,...,varn;
  prompt "Update row?";
  exit when answer = "yes";
end loop

set selection variables of cursor C2 to re-select the row to be updated;

exec sql START;
exec sql OPEN C2;
exec sql FETCH C2 INTO :v1r1,:var2,...,:varn;
exec sql UPDATE table SET ... WHERE CURRENT OF C2;
exec sql COMMIT;
```

Here, all FETCH statements in the loop are excluded from the transaction handling. By using this mechanism only the FETCH against cursor C2 is in the read-set of the update transaction, minimizing the risk of a conflict.

6.1.2 Locking

In any database system, at some stage in a transaction, the data records must be locked to prevent access by other processes and to ensure that the transaction is not interrupted. In the MIMER system, no change is made to the database contents during the transaction build-up and no records are locked. This means that the database can be freely accessed (and updated) by any other process; the data accessed by the transaction is only locked during the commit phase. In this way, locks are held only for a very short interval of time. The problems associated with locking are further reduced since only those records that are actually to be updated are locked. Other data in the same table continues to be accessible to other transactions.

Deadlock situations, which can be relatively common in database management systems where records are locked during transaction build-up, can not occur in MIMER. In MIMER it is impossible for two processes to be waiting for a record locked by the other process. In some database management systems this situation may require operator intervention to be resolved.

6.2 Transaction control statements

6.2.1 Starting transactions

Transaction start may be set to EXPLICIT or IMPLICIT.

The default transaction start setting is IMPLICIT, which means a transaction will be started **automatically** whenever an update is performed.

To set the transaction start mode, use the statements:

```
SET TRANSACTION START EXPLICIT;  
SET TRANSACTION START IMPLICIT;
```

Different database connections may use different transaction start options.

The START statement can always be used to explicitly start a transaction. This is useful if a number of related updates are to be performed, and it is desirable that all the updates succeed or fail together to maintain consistency.

You cannot start a transaction while a transaction is already active.

Explicit transaction start

With this setting, transactions are never automatically started. All transactions **must** be explicitly started by issuing a START statement.

Any update operation (insert, update, delete) involving a table in a databank with the **TRANS** or **LOG** option must occur within a transaction. An error will be raised if such an update is attempted without first starting a transaction.

All the statements issued after the START statement and before the COMMIT or ROLLBACK statement are grouped together within that single transaction.

A transaction is concluded by issuing a COMMIT or ROLLBACK statement.

Implicit transaction start

With this setting, a transaction is started **automatically** by an update, regardless of the databank option(s) of affected databank(s).

The START statement may be used to explicitly start a transaction if required, typically to allow several updates to grouped together within a single transaction for consistency, as already described.

A transaction is concluded by issuing a COMMIT or ROLLBACK statement.

All the statements issued after the initiating update and before the concluding COMMIT or ROLLBACK statement are grouped together within that single transaction.

Note: In version 5 of MIMER/SQL, the default setting was START EXPLICIT. This is still true for programs which use the version 5 syntax for CONNECT. The version 7 CONNECT TO statement sets the default transaction setting to IMPLICIT.

6.2.2 Ending transactions

Transactions must be ended with the COMMIT or ROLLBACK statement.

COMMIT requests that the operations in the write-set are executed on the database making the changes permanent and visible to other users. The SQLSTATE value returned from a COMMIT statement indicates either that the transaction commitment was successful (SQLSTATE = '00000') or that a transaction conflict occurred (SQLSTATE \neq '00000').

ROLLBACK abandons the transaction. The read- and write-sets are dropped, and no changes are made to the database. ROLLBACK is always successful.

Note. The keyword ROLLBACK is used in MIMER/SQL for compatibility with SQL standards. A transaction in MIMER is never physically 'rolled back' in the sense of undoing changes made to the database, since changes are not actually effected until a successful COMMIT is performed.

Transactions which are not successfully committed due to a transaction conflict do not have to be explicitly rolled back. The ROLLBACK statement is most commonly used in exception routines for handling error situations which might arise during transaction build-up.

If a connection or program is terminated without requesting a COMMIT or ROLLBACK for the current transaction, the system will abort the transaction. No changes requested during the transaction build-up will be made to the database.

Transaction handling in ISQL and BSQL differs slightly from that described here - see Section 6.3.1 of the MIMER/SQL Interactive User's Manual.

6.2.3 Read-through-write-set

To ensure complete consistency in the information seen by an application, a transaction should strictly check its own write-set to see whether data accessed at one point in the transaction build-up was modified by an update request earlier in the same transaction. This can be relatively time-consuming for large transactions. The `SET TRANSACTION CHANGES` statement determines whether read-through-write-set is active:

INVISIBLE is the default setting; a transaction does not read through its own write-set during build-up. If the transaction includes multiple changes to the same data, the information seen by the user may not be consistent with the database contents.

VISIBLE activates read-through-write-set, ensuring complete consistency in all transactions. Use this setting for transactions which make multiple changes in the same transaction.

It is generally preferable to divide multiple changes into several transactions, rather than using `SET TRANSACTION CHANGES VISIBLE`, since read-through-write-set can reduce performance significantly for large transactions. The **VISIBLE** setting should only be used where multiple changes in the same transaction are unavoidable.

6.2.4 Statements allowed in transactions

The table below summarizes which statements may be used inside transactions.

Statement	Allowed in transactions	Comments
<i>Access control statements</i>		
CONNECT, SET CONNECTION	Yes	
DISCONNECT, ENTER, LEAVE	No	
<i>Data manipulation statements</i>		
DECLARE CURSOR, WHENEVER	Irrelevant	Declarative statement
SELECT INTO, FETCH, INSERT, DELETE, UPDATE	Yes	
OPEN, CLOSE	Yes	COMMIT and ROLLBACK close any open cursors
PREPARE, DESCRIBE, EXECUTE, ALLOCATE, DEALLOCATE, GET DESCRIPTOR, SET DESCRIPTOR	Yes	Dynamic statements (see Chapter 7)
<i>Data definition statements</i>		
CREATE, ALTER, COMMENT, DROP, GRANT, REVOKE	No	These statements create internal transactions to ensure data dictionary consistency
<i>Diagnostic statements</i>		
GET DIAGNOSTICS	Yes	
<i>System administration statements</i>		
ALTER DATABANK RESTORE, CREATE BACKUP, CREATE INCREMENTAL BACKUP, SET DATABANK, SET DATABASE, SET SHADOW, UPDATE STATISTICS	No	These statements create internal transactions to ensure data dictionary consistency

6.2.5 Cursors in transactions

The transaction terminating statements `COMMIT` and `ROLLBACK` automatically close any cursors opened by the current connection. If a cursor is stacked, all instances of the cursor are closed. Any cursors which may be retained in dormant connections are not affected. Cursors are also closed automatically by `LEAVE` and `DISCONNECT`.

In the `SET TRANSACTION START EXPLICIT` mode, cursors may be opened and used outside transactions. Such cursors remain open when an `ENTER` or `LEAVE` statement is issued. This is illustrated in the following statement sequence:

```
...
exec sql SET TRANSACTION START EXPLICIT;
exec sql DECLARE C1 CURSOR FOR SELECT col1 FROM tab1;
exec sql DECLARE C2 CURSOR FOR SELECT col2 FROM tab2
                                WHERE checkcol = :var1;

exec sql OPEN C1;
loop
    exec sql FETCH C1 INTO :var1;      -- fetch value from tab1
    exec sql ENTER ... ;              -- change current ident
        exec sql OPEN C2;
        exec sql FETCH C2 INTO ...;    -- fetch row for C2
        exec sql CLOSE C2;
    exec sql LEAVE;
end loop;
...
```

In the above example, the value fetched for the cursor `C1` is used to determine the set of rows addressed by cursor `C2`. Cursor `C1` remains open and positioned during the `ENTER...LEAVE` sequence. Each time the loop is executed, a new value is fetched by `C1` and a new set of its rows addressed by `C2`. The same behaviour applies when `LEAVE RETAIN` is used to leave a program ident but keep the environment for the ident.

A cursor opened and used outside a transaction may however not be used within a transaction. If the same cursor is required outside and inside a transaction, separate instances must be opened. Remember that separate instances of a cursor address separate result tables:

```
...
exec sql SET TRANSACTION START EXPLICIT;
exec sql DECLARE C1 REOPENABLE CURSOR FOR SELECT col1 FROM tab1;
exec sql OPEN C1;
exec sql FETCH C1 INTO ...;          -- first row (outside transaction)
...
exec sql START;
exec sql OPEN C1;                    -- new instance of cursor
exec sql FETCH C1 INTO ...;          -- first row again
...
```

6.2.6 Error handling in transactions

In general, errors and exception conditions are reported in SQLSTATE after each executable SQL statement. The value of SQLSTATE indicates the outcome of the preceding statement (see Appendix C). GET DIAGNOSTICS can be used to get detailed status information after an SQL statement.

The value of SQLSTATE after a COMMIT statement indicates the success or failure of the request to commit the transaction, not the outcome of any data manipulation performed in the transaction.

Use of the general error handling statement WHENEVER (see MIMER/SQL Reference Manual, 'SQL statement descriptions', for a description of WHENEVER) in transactions requires some care:

- Program control can be transferred to an exception routine in the event of an error. Make sure that the exception routine is designed to take care of uncompleted transactions. Most commonly, the first SQL statement in the exception routine should be GET DIAGNOSTICS. The exception routine should normally also execute a ROLLBACK statement. Remember that if the exception routine is used from a statement outside a transaction, any open cursors belonging to the current ident will be closed by the ROLLBACK statement.
- For transaction conflict, the SQLSTATE value returned from the COMMIT statement falls into the SQLERROR class. If the transaction is to be retried in the event of conflict, make sure that no 'WHENEVER SQLERROR GOTO exception' statement is operative. If WHENEVER error handling is used in an application program, a suitable program structure for COMMIT statements is

```
exec sql WHENEVER SQLERROR GOTO exception;
...
exec sql WHENEVER SQLERROR GOTO retry;
exec sql COMMIT;
exec sql WHENEVER SQLERROR GOTO exception;
...
```

6.3 Transactions and logging

Changes made to a database may be logged, to provide back-up protection in the event of hardware failure, provided that the changes are included in a transaction and that the databanks concerned have LOG option. Transaction handling is thus important even in standalone environments where concurrency control requirements do not arise.

Transaction and logging requirements are controlled at the databank level by the option (LOG, TRANS or NULL) set for the databank. Only operations on LOG databanks are logged. Write operations against tables in LOG and TRANS databanks must be performed within a transaction.

6.4 Protection against data loss

6.4.1 System interruptions

If a transaction build-up is interrupted by a system failure or a program termination (deliberate or otherwise) the transaction is aborted, none of the requested changes are made on the database.

Transactions which are interrupted after the request to commit but before all operations in the transaction have been executed on the database are completed by the automatic recovery functionality when the databank in question is next accessed. There is no possibility of transaction conflict in such an automatic completion, since no other process can access the data in question as long as an incomplete transaction is pending.

In the event of a system failure which interrupts one or more application programs, it may be necessary to examine the database contents 'by hand' to determine which transactions failed to commit before the interruption.

6.4.2 Hardware failure

A databank which is damaged by hardware failure (e.g. a disk crash) may be recovered using back-up copies and the databank log, provided that all write operations on the databank have been logged (and that the log databank and back-up copies are intact). The backup and restore facilities are described in the System Management Handbook.

7 DYNAMIC SQL

7.1 Principles of dynamic SQL

Dynamic SQL allows you to execute SQL statements placed in a string variable instead of explicitly writing the statements inside a program. This allows SQL statements to be constructed within an application program. These facilities are typically used in interactive environments, where SQL statements are submitted to the application program from the terminal.

Use of dynamic SQL has been significantly simplified in version 7.3 of MIMER/SQL, since the standardized way to handle SQL descriptor areas has been implemented.

An example of when dynamic SQL is needed would be a program for interactive SQL, where any correct SQL statement may be entered at the terminal and processed by the application. Limited dynamic facilities may however be provided by relatively simple application programs.

The following classes of SQL statements may be submitted to programs using dynamic SQL. Statements excluded from dynamic applications are declarations, diagnostic statements and dynamic SQL statements themselves.

Access control statements:

ENTER
LEAVE

Data definition statements:

CREATE
ALTER
COMMENT
DROP

Security control statements:

GRANT
REVOKE

Transaction control statements:

SET TRANSACTION
START
COMMIT
ROLLBACK

Data manipulation statements:

```
SELECT
INSERT
UPDATE
UPDATE CURRENT
DELETE
DELETE CURRENT
```

System administration statements:

```
CREATE BACKUP
CREATE INCREMENTAL BACKUP
ALTER DATABANK RESTORE
SET DATABASE
SET DATABANK
SET SHADOW
UPDATE STATISTICS
```

Statements may be submitted to dynamic SQL applications in two forms:

- Fully defined statements, written exactly as they would be submitted to interactive SQL. For example:

```
GRANT ALL ON HOTEL TO CHARLIE

UPDATE ROOM_STATUS SET STATUS = 'KEY OUT'
WHERE ROOMNO = 'SKY112'

SELECT * FROM HOTEL

SELECT RESERVATION, SUM(AMOUNT)
FROM BILL
GROUP BY RESERVATION
```

- Statements with 'parameter markers', which identify positions where the value of a host variable will be inserted when the statement is executed or the cursor is opened. A parameter marker is represented by a question mark. For example:

```
UPDATE ROOM_STATUS SET STATUS = ? WHERE ROOMNO = ?

DELETE FROM BOOK_GUEST WHERE RESERVATION = ?

SELECT HOTELCODE, ON_DATE, FREECOUNT*?
FROM FREEROOMS
WHERE ROOMTYPE = ?
AND HOTELCODE IN (SELECT HOTELCODE
FROM HOTEL
WHERE CITY = ?)
```

Statements submitted with parameter markers are equivalent to normal embedded statements using host variables, except that the statements are defined at run-time.

7.2 General summary of dynamic SQL processing

The following statements are used when SQL statements are dynamically submitted:

<u>Statement</u>	<u>Description</u>
ALLOCATE CURSOR	Allocate extended cursor.
ALLOCATE DESCRIPTOR	Allocate SQL descriptor area.
CLOSE	Close an open cursor.
DEALLOCATE DESCRIPTOR	Deallocate SQL descriptor area.
DEALLOCATE PREPARE	Deallocate prepared SQL statement.
DECLARE CURSOR	Declare a cursor for a statement which will be dynamically submitted.
DESCRIBE	Examine the object form of the statement, and assigns values to the appropriate parameters in the SQL descriptor area.
EXECUTE	Execute a prepared statement (except SELECT statements).
EXECUTE IMMEDIATE	Shorthand form for PREPARE followed by EXECUTE. This form can only be used for fully-defined non-SELECT statements with no parameter markers.
FETCH	Fetch rows for a dynamic cursor.
GET DESCRIPTOR	Get values from the SQL descriptor area.
OPEN	Open a prepared cursor.
PREPARE	Compile an SQL source statement into an internal object form.
SET DESCRIPTOR	Set values in the SQL descriptor area.

All statements submitted to dynamic SQL programs must be prepared.

All prepared statements except SELECT statements are executed with the EXECUTE statement. A SELECT statement is 'executed' by the use of OPEN and FETCH for a cursor declared with the prepared SELECT statement.

The declaration of a cursor for a statement (DECLARE CURSOR) must always precede the PREPARE operation for the same statement in an application using dynamic SQL.

7.3 SQL descriptor area

The SQL descriptor area is used for managing variable references in dynamically submitted SQL statements where the number and/or data type of the host variables required is not known at the time when the program is written. An SQL descriptor area is allocated with the embedded SQL statement `ALLOCATE DESCRIPTOR` and deallocated with `DEALLOCATE DESCRIPTOR`. See the MIMER/SQL Reference Manual, 'SQL statement descriptions', for a description of these SQL statements. A program may allocate several separate descriptor areas, identified by different descriptor names. SQL descriptor areas are used for both `SELECT` list items and input host variables.

The following statement types can use information from SQL descriptor areas:

- all `SELECT` statements
- `INSERT`, `DELETE` and `UPDATE` statements using parameter markers
- `ENTER` statements

The following statement types cannot use SQL descriptor areas:

- all data definition statements, security control statements, access control statements (except `ENTER`) and transaction control statements
- `INSERT`, `DELETE` and `UPDATE` statements using constant values

In practice, programs using dynamically submitted SQL statements are usually written as though all submitted statements use SQL descriptor areas (since the nature of the submitted statement is not known until run-time). SQL descriptor areas can be left out of a program only if it is known in advance that they will not be needed (for instance in an application program which will handle only submitted data definition statements).

7.3.1 The structure of the SQL descriptor area

The SQL descriptor area is a storage area holding information about the described statement. It is allocated and maintained with embedded SQL statements. It consists of one `COUNT` field and one or more item descriptor areas, each one describing either a selected column or an input/output value:

COUNT
item descriptor area 1
item descriptor area 2
...
item descriptor area n

The `COUNT` field specifies how many of the item descriptor areas contain data.

7.3.2 The item descriptor area fields

Each item descriptor area describes a selected column, an input variable or an output variable. In all cases but DESCRIBE the item descriptor area contains a value, and a description of that value. In DESCRIBE, the item descriptor area instead describes a column of the result-set of a select-cursor specification.

The individual fields of the item descriptor area can be accessed with the GET DESCRIPTOR and SET DESCRIPTOR statements. See the MIMER/SQL Reference Manual for a description of these statements.

Observe that not all fields are relevant for all descriptor areas. Which fields are used depends on the data type and other attributes.

The item descriptor area contains the following fields:

TYPE	An exact numeric value with scale 0, containing a coded representation of the data type. See below for a description of the data type codes.
LENGTH	An exact numeric value with scale 0, containing the character string length of a character string data type. Terminating null bytes are excluded.
PRECISION	An exact numeric value with scale 0, specifying the precision for a numeric data type value. For the data types INTERVAL DAY TO SECOND, INTERVAL HOUR TO SECOND, INTERVAL MINUTE TO SECOND, INTERVAL SECOND, TIME and TIMESTAMP the value in this field describes the precision of the fractional SECONDS component.
SCALE	An exact numeric value with scale 0, specifying the scale for a numeric data type value.
NULLABLE	An exact numeric value with scale 0, indicating whether a resulting column can contain NULL or not. NULLABLE=1 indicates that NULL is allowed. NULLABLE=0 indicates that NULL is not allowed.

INDICATOR	An exact numeric value with scale 0, used as a NULL indicator for input (OPEN or EXECUTE) or output (FETCH) values. INDICATOR=-1 indicates a NULL value, and INDICATOR=0 indicates a non-NULL value. If INDICATOR is > 0 after a FETCH operation it indicates that a truncation occurred, and the value of INDICATOR is the required length.
DATA	If the INDICATOR field does not indicate a NULL value, this field contains an input (OPEN or EXECUTE) or output (FETCH) value with the data type specified by the TYPE field, and with the attributes specified by applicable fields in the item descriptor area.
NAME	A character string containing the column name, returned by DESCRIBE OUTPUT. After DESCRIBE INPUT this field contains a question mark ('?').
UNNAMED	An exact numeric value with scale 0, indicating whether NAME contains an actual column name or a situation dependent generated name (for example if the value is a result of an aggregate function). UNNAMED=0 indicates that NAME contains an actual column name. UNNAMED=1 means that NAME does not contain an actual column name.
RETURNED_LENGTH	An exact numeric value with scale 0, set by FETCH, which returns the actual character length of a VARCHAR output value, or of a VARCHAR result column value if the datatype of the output is not VARCHAR.

DATETIME_INTERVAL_CODE	If the TYPE field contains 9 or 10 (i.e. for DATETIME and INTERVAL data types), this column will contain an exact numeric value with scale 0 which specifies the DATETIME or INTERVAL subtype. See below for descriptions of the codes that apply to these two data types.
DATETIME_INTERVAL_PRECISION	An exact numeric value with scale 0, which specifies the leading field precision for the INTERVAL data type (i.e. when the TYPE value is 10).

Before the DATA field is read by GET DESCRIPTOR or set by SET DESCRIPTOR, the TYPE field and the other applicable fields must be set to consistent and appropriate values. The data type of a host variable receiving or supplying the DATA field must match the TYPE field. The rules for host variable data type conversion, described in the MIMER/SQL Reference Manual 'Basic SQL syntax rules', are followed for input and output values.

7.3.3 SQL data type codes

The SQL data type for a value or a select column is specified with a numeric code in the TYPE field of the item descriptor area. Note that some of the codes are negative. This is because the SQL standard states that implementation defined codes should be negative.

The TYPE field in the item descriptor area can contain one of the following values:

Code	SQL data type
1	CHARACTER
2	NUMERIC
3	DECIMAL
4	INTEGER
5	SMALLINT
6	FLOAT
7	REAL
8	DOUBLE PRECISION
9	DATETIME
10	INTERVAL
12	CHARACTER VARYING
-11	INTEGER(p)*
-12	FLOAT(p)*
-13	CHARACTER VARYING**

* INTEGER(p) and FLOAT(p) are MIMER-specific data types used for integer and float data with a specified precision.

** The MIMER-specific type code -13 is used by application programs when storing binary (see Appendix F.5 for an example).

For DATETIME data types, the DATETIME_INTERVAL_CODE field in the item descriptor area can contain one of the following values:

Code	DATETIME subtype
1	DATE
2	TIME
3	TIMESTAMP

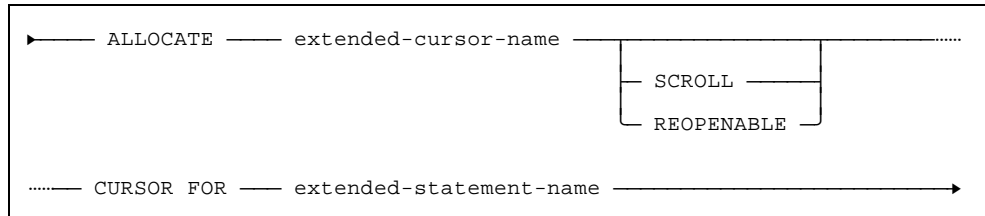
For INTERVAL data types, the DATETIME_INTERVAL_CODE field in the item descriptor area can contain one of the following values:

Code	INTERVAL subtype
1	YEAR
2	MONTH
3	DAY
4	HOUR
5	MINUTE
6	SECOND
7	YEAR TO MONTH
8	DAY TO HOUR
9	DAY TO MINUTE
10	DAY TO SECOND
11	HOUR TO MINUTE
12	HOUR TO SECOND
13	MINUTE TO SECOND

7.5 Extended dynamic cursors

A "normal" cursor is identified by an SQL identifier. An extended cursor makes it possible to represent a dynamic cursor by a host variable or a literal. An extended cursor is allocated by the application with the ALLOCATE CURSOR statement.

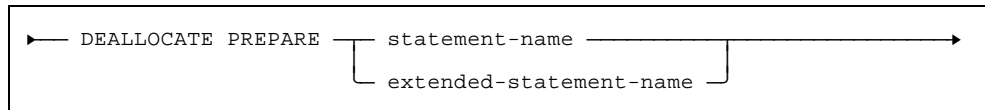
This statement has the following syntax:



The ALLOCATE CURSOR statement is further described in the 'MIMER/SQL Reference Manual'.

The *extended-statement-name* differs from a "normal" statement in the same way as the extended cursor differs from a "normal" cursor; it is identified by a host variable or a literal, instead of by an identifier. The statement must have been prepared by a PREPARE statement earlier in the application.

When the application is finished with the processing of the SQL statement, the prepared statement may be destroyed with a DEALLOCATE PREPARE call. DEALLOCATE PREPARE also destroys any extended cursor that was associated with the statement. DEALLOCATE PREPARE has the following syntax:



Example of how extended cursors are used:

```

...
exec sql BEGIN DECLARE SECTION;
SQL_TXT CHARACTER(255);
C1 CHARACTER(18);
STM1 CHARACTER(18);
ITEM INTEGER;
...
exec sql END DECLARE SECTION;
...
SQL_TXT := 'SELECT COL1,COL2 FROM TAB1';
STM1 := "STATE_1";
exec sql PREPARE :STM1 FROM :SQL_TXT;
C1 := "CUR1";
exec sql ALLOCATE :C1 CURSOR FOR :STM1;
...
exec sql ALLOCATE DESCRIPTOR 'SQLA' WITH MAX 50;
exec sql DESCRIBE OUTPUT :STM1 USING SQL DESCRIPTOR 'SQLA';
...
exec sql OPEN :C1;
exec sql WHENEVER NOT FOUND GOTO done;
  
```

```

loop
  exec sql FETCH :C1 INTO SQL DESCRIPTOR 'SQLA';
  exec sql GET DESCRIPTOR 'SQLA' VALUE 1 :HOSTVAR1 = DATA;
  exec sql GET DESCRIPTOR 'SQLA' VALUE 2 :HOSTVAR2 = DATA;
  ...
  display HOSTVAR1,HOSTVAR2,...;
end loop;
done:
exec sql CLOSE :C1;
exec sql DEALLOCATE DESCRIPTOR 'SQLA';
exec sql DEALLOCATE PREPARE :STM1;
...

```

7.6 Describing prepared statements

SELECT statements and statements containing parameter markers can be described to obtain information about the number and data types of the parameters. There are two forms of DESCRIBE:

- DESCRIBE OUTPUT for SELECT-list parameters
- DESCRIBE INPUT for input parameters.

Both forms of DESCRIBE use the object (prepared) form of the statement as an argument. The same statement may be described in both senses if necessary.

For example:

```

exec sql BEGIN DECLARE SECTION;
SQLA1  CHARACTER(18);
MAXOCC INTEGER;
SOURCE CHARACTER(255);
exec sql END DECLARE SECTION;
...
MAXOCC := 15;
SQLA1 := "SQL_AREA_1";
exec sql ALLOCATE DESCRIPTOR :SQLA1 WITH MAX 20;
exec sql ALLOCATE DESCRIPTOR 'SQLA2' WITH MAX :MAXOCC;
...
exec sql PREPARE OBJECT FROM :SOURCE;
exec sql DESCRIBE OUTPUT OBJECT USING SQL DESCRIPTOR :SQLA1;
exec sql DESCRIBE INPUT OBJECT USING SQL DESCRIPTOR 'SQLA2';
...

```

DESCRIBE places information about the prepared statement in the SQL descriptor areas. See Section 7.3 for a description of the SQL descriptor area. The contents of the SQL descriptor area is read with the GET DESCRIPTOR statement, and updated with the SET DESCRIPTOR statement.

7.6.1 Describing SELECT-lists

Items in the SELECT list of a SELECT statement are described with DESCRIBE OUTPUT. The keyword OUTPUT may be omitted.

The result of the description with DESCRIBE OUTPUT indicates

- Whether the statement is a SELECT statement or not. This is indicated by the COUNT field, which is set to zero for non-SELECT statements. Dynamic SQL programs should test for this condition after each DESCRIBE operation, since the execution of SELECT and non-SELECT statements differs (see Section 7.7). If the statement is a SELECT statement, DESCRIBE will place information about the SELECT list in the fields of the item descriptor areas.
- Whether the current descriptor area allocation is sufficient or not. Insufficient area is indicated by the SQLSTATE variable set to a warning state, and a value of COUNT (actual number of items) greater than that specified in the 'WITH MAX ...'-clause of the ALLOCATE DESCRIPTOR statement, or greater than 100 if no 'WITH MAX...'-clause is specified. If the area is insufficient, no items are described.

7.6.2 Describing input variables

Parameter markers are described with the DESCRIBE INPUT statement.

The value of the COUNT field indicates the number of parameter markers in the statement; a value of zero indicates no input parameters. (Note the functional difference from DESCRIBE OUTPUT, where a value of zero in the COUNT field indicates that the statement is not a SELECT statement). A value greater than that specified in 'WITH MAX...' indicates that the allocated SQL descriptor area is too small, and the description cannot be performed. This situation is handled as described above for DESCRIBE OUTPUT.

7.7 Executing statements

After PREPARE and DESCRIBE, the way in which submitted statements are handled differs according to whether the statement is SELECT or not:

- Non-SELECT** statements are executed using the EXECUTE statement, with the object (prepared) form of the submitted statement as the argument.
- SELECT** statements use a cursor, associated with the object form of the prepared statement, and are 'executed' with OPEN and FETCH. SELECT statements may not be processed with EXECUTE.

7.7.1 Non-SELECT statements

Non-SELECT statements are identified by a value of zero in the COUNT field of the SQL descriptor area after DESCRIBE OUTPUT, or by a value other than 'SELECT' in the DYNAMIC_FUNCTION fields in the diagnostics area, retrieved by the GET DIAGNOSTICS statement. If the statement does not contain user variables, it may be executed directly. If on the other hand the source form of the statement contains parameter markers, the statement is executed using the SQL descriptor area containing the marker description. The pseudo code uses the logical variables SELECT and USERVAR to flag the type of statement and presence of user variables respectively:

```

...
exec sql ALLOCATE DESCRIPTOR 'SQLA1' WITH MAX 30;
exec sql ALLOCATE DESCRIPTOR 'SQLA2' WITH MAX 30;
...
exec sql DESCRIBE OUTPUT OBJECT USING SQL DESCRIPTOR 'SQLA1';
exec sql GET DESCRIPTOR 'SQLA1' :NO_OUT = COUNT;
if NO_OUT = 0 then
    SELECT := FALSE;
else
    SELECT := TRUE;
end if;
...
exec sql DESCRIBE INPUT OBJECT USING SQL DESCRIPTOR 'SQLA2';
exec sql GET DESCRIPTOR 'SQLA2' :NO_IN = COUNT;
if NO_IN = 0 then
    USERVAR := FALSE;
else
    USERVAR := TRUE;
end if;
...
if SELECT then
    ...
else
    if USERVAR then
        exec sql EXECUTE OBJECT USING SQL DESCRIPTOR 'SQLA2';
    else
        exec sql EXECUTE OBJECT;
    end if;
...

```

The descriptor area referenced in the EXECUTE statement may be replaced by an explicit list of host variables, provided that the number and data types of the user variables in the source statement are known when the program is written (so that variables can be declared and the appropriate variable list written into the EXECUTE statement). This facility is of limited use, since the occasions when the user constructs freely chosen SQL statements with a predetermined number of user variables are rare.

The shorthand form EXECUTE IMMEDIATE combines the functions of PREPARE and EXECUTE. This form may only be used for non-SELECT statements with no parameter markers, and is therefore of value only in contexts where the user is restricted to this type of statement. (Data definition and security control statements fall into this category, since user variables are not permitted in the syntax of these statements. EXECUTE IMMEDIATE can therefore be useful for application programs designed specifically to handle database definition statements).

7.7.2 SELECT statements

SELECT statements are identified by a non-zero value in the COUNT field of the SQL descriptor area after DESCRIBE OUTPUT, or by the value 'SELECT' in the DYNAMIC_FUNCTION fields in the diagnostics area, retrieved by a GET DIAGNOSTICS statement. All dynamically submitted SELECT statements must be handled through cursors. The special SELECT INTO statement for single-row selection directly into host variables may not be submitted to dynamic SQL applications.

Cursors are declared or allocated for the object (prepared) form of submitted SELECT statements. Note that a DECLARE CURSOR statement must precede the preparation of the SELECT statement. If ALLOCATE CURSOR is used instead of DECLARE CURSOR, it must be preceded by the PREPARE statement. The SQL statement must also be prepared before the cursor is opened.

If the source form of the SELECT statement contains parameter markers, these must be described before the cursor is opened, and the OPEN statement must reference the relevant descriptor area. In the rare case where the number and data type of the user variables are known when the program is first written, the OPEN statement may reference an explicit variable list instead of a descriptor area.

The descriptor area used for the SELECT list of the submitted statement is referenced when data is retrieved with the FETCH statement.

See the following example:

```
...
exec sql DECLARE C1 CURSOR FOR OBJECT;
exec sql ALLOCATE DESCRIPTOR 'SQLA1' WITH MAX 30;
exec sql ALLOCATE DESCRIPTOR 'SQLA2' WITH MAX 30;
...
exec sql PREPARE OBJECT FROM :SOURCE;
...
exec sql DESCRIBE OUTPUT OBJECT USING SQL DESCRIPTOR 'SQLA1';
exec sql GET DESCRIPTOR 'SQLA1' :NO_OUT = COUNT;
```

```

if NO_OUT = 0 then
    SELECT := FALSE;
else
    SELECT := TRUE;
end if;
...
exec sql DESCRIBE INPUT OBJECT USING SQL DESCRIPTOR 'SQLA2';
exec sql GET DESCRIPTOR 'SQLA2' :NO_IN = COUNT;
if NO_IN = 0 then
    USERVAR := FALSE;
else
    USERVAR := TRUE;
end if;
...
if SELECT then
    if USERVAR then
        exec sql OPEN C1 USING SQL DESCRIPTOR 'SQLA2';
    else
        exec sql OPEN C1;
    end if;
    ...
    exec sql FETCH C1 INTO SQL DESCRIPTOR 'SQLA1';
    ...
else
    ...
end if;

```

7.8 Example framework for dynamic SQL programs

This section gives a general framework (in pseudo code) for dynamic SQL programs designed to handle any valid SQL statement as input. The framework is largely a synthesis of the example fragments given earlier in this chapter.

The framework is written as a single sequential module to emphasise the order of operations.

Host variable declarations are omitted. Handling of values returned by FETCH is also omitted.

See Appendix F for a complete example of how to use dynamic SQL in the C language.

Example framework (features specific to real host languages are described in Appendix A):

```

...
--
-- Allocate 2 descriptor areas
--
exec sql ALLOCATE DESCRIPTOR 'SQLA1' WITH MAX 50;
exec sql ALLOCATE DESCRIPTOR 'SQLA2' WITH MAX 50;
--
-- declare cursor for SELECT statements
--
exec sql DECLARE C1 CURSOR FOR OBJECT;
--
-- read statement from terminal
--
read INPUT into SOURCE;
--
-- prepare statement
--
exec sql PREPARE OBJECT FROM :SOURCE;
--
-- describe statement and set type/parameter usage flags
--
exec sql DESCRIBE OUTPUT OBJECT USING SQL DESCRIPTOR 'SQLA1';
exec sql GET DESCRIPTOR 'SQLA1' :NO_OUT = COUNT;
if NO_OUT = 0 then
    SELECT := FALSE;
else
    SELECT := TRUE;
end if;
--
exec sql DESCRIBE INPUT OBJECT USING 'SQLA2';
exec sql GET DESCRIPTOR 'SQLA2' :NO_IN = COUNT;
if NO_IN = 0 then
    USERVAR := FALSE;
else
    USERVAR := TRUE;
end if;
--
-- execute statement or open cursor and fetch after assigning
-- values to input variables
--
if SELECT then
    if USERVAR then
        exec sql OPEN C1 USING SQL DESCRIPTOR 'SQLA2';
    else
        exec sql OPEN C1;
    end if;
    loop
        exec sql FETCH C1 INTO SQL DESCRIPTOR 'SQLA1';
        exit when NO_MORE_REQUIRED or SQLSTATE = "02000";
        ... -- process results of FETCH
    end loop;
    exec sql CLOSE C1;
else
    if USERVAR then
        exec sql EXECUTE OBJECT USING SQL DESCRIPTOR 'SQLA2';
    else
        exec sql EXECUTE OBJECT;
    end if;
end if;
...

```


8 HANDLING ERRORS AND EXCEPTION CONDITIONS

Errors may arise at three general levels in an embedded SQL program (not counting errors in the SQL-independent host language code). These are syntax, semantic and run-time errors.

8.1 Syntax errors

Syntax errors are constructions which break the rules for formulating SQL statements. For example:

- spelling errors in keywords

```
SLEECT                                for    SELECT
```

- incorrect or missing delimiters

```
DELETEFROM                            for    DELETE FROM  
SELECT column1;column2                for    SELECT column1,column2
```

- incorrect clause ordering

```
UPDATE ..WHERE..SET                    for    UPDATE..SET..WHERE
```

Syntactically incorrect statements are not accepted by the preprocessor. The error must be corrected before the program can be successfully preprocessed.

8.2 Semantic errors

Semantic errors arise when SQL statements are formulated in full accordance with the syntax rules, but do not reflect the programmer's intentions correctly. Some semantic errors, notably incorrect references to database objects, are detected by the SQL compiler.

Semantic errors detected during preprocessing give warnings, without aborting the preprocessor.

8.3 Run-time errors

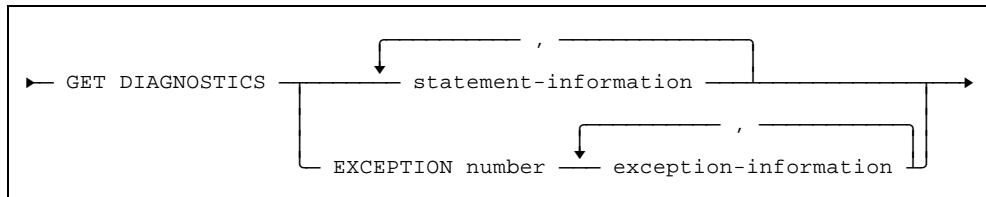
Run-time errors and exception conditions (for example warnings) arising during execution of embedded SQL statements are signalled by the contents of the `SQLSTATE` status variable described in Section 3.2. A list of possible `SQLSTATE` values is provided in Appendix C. The `GET DIAGNOSTICS` statement can be used to retrieve detailed information about an exception.

8.3.1 GET DIAGNOSTICS

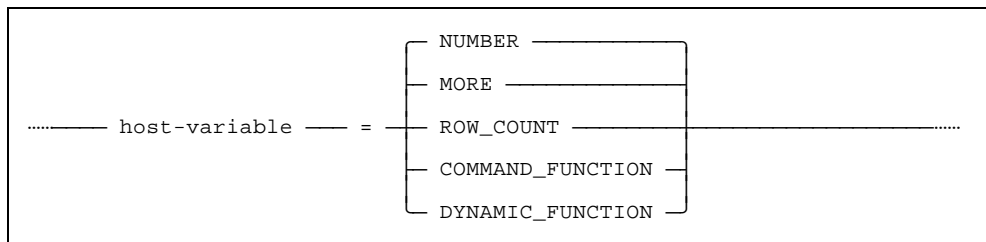
The GET DIAGNOSTICS statement retrieves selected status information from the diagnostics area. The diagnostics area holds information about the most recently executed SQL statement. Observe that the GET DIAGNOSTICS statement itself does not change the diagnostics area.

The GET DIAGNOSTICS statement can be in two forms: The first form retrieves statement information about the most recently SQL statement executed, if it was successful. The second form of GET DIAGNOSTICS is the "EXCEPTION" form, which retrieves exception information if the most recently executed SQL statement failed. The ordinal number of the exception to be returned is specified by *number* (see below).

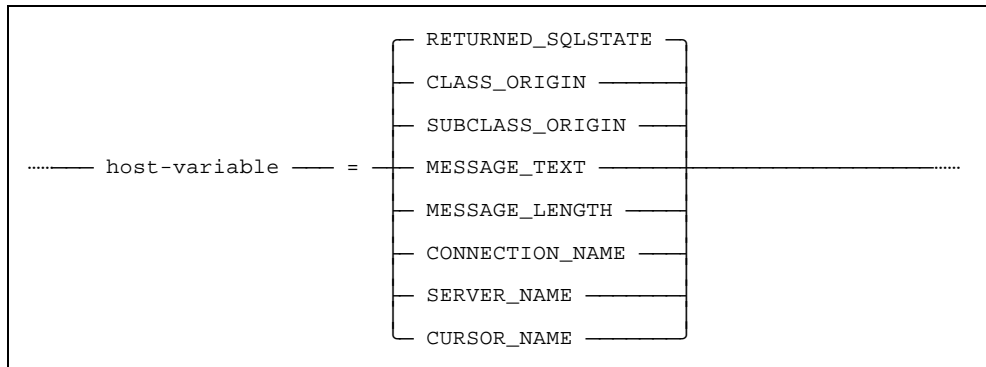
GET DIAGNOSTICS has the following syntax:



where *statement-information* is:



and *exception-information* is:



The following information items can be retrieved from the diagnostics area:

<u>Information item</u>	<u>Data type</u>	<u>Description</u>
NUMBER	INTEGER	The number of exception messages stored for the most recently executed SQL statement.
MORE	CHAR(1)	Tells if there are any exceptions for which no exception information has been stored.
ROW_COUNT	INTEGER	The number of rows inserted, updated or deleted if the last statement was INSERT, searched UPDATE or DELETE.
COMMAND_FUNCTION	VARCHAR(128)	A string identifying the preceding embedded SQL statement executed.
DYNAMIC_FUNCTION	VARCHAR(128)	A string identifying the preceding prepared SQL statement executed.
RETURNED_SQLSTATE	CHAR(5)	Value of SQLSTATE for the specified exception.
CLASS_ORIGIN	VARCHAR(128)	The defining source of the two first characters (the class portion) of the SQLSTATE value.
SUBCLASS_ORIGIN	VARCHAR(128)	The defining source of the three last characters (the subclass portion) of the SQLSTATE value.
MESSAGE_TEXT	VARCHAR(254)	A descriptive message text for the specified exception.
MESSAGE_LENGTH	INTEGER	The length of the message text for the specified exception.
CONNECTION_NAME	VARCHAR(128)	The connection name specified in a CONNECT, DISCONNECT or SET CONNECTION statement. The name of the current connection for all other statements.
SERVER_NAME	VARCHAR(128)	The database name specified by a CONNECT, DISCONNECT or SET CONNECTION statement. The current database name for all other statements.
CURSOR_NAME	VARCHAR(128)	The cursor name associated with the statement.

The `COMMAND_FUNCTION` and `DYNAMIC_FUNCTION` information items can contain any of the following values:

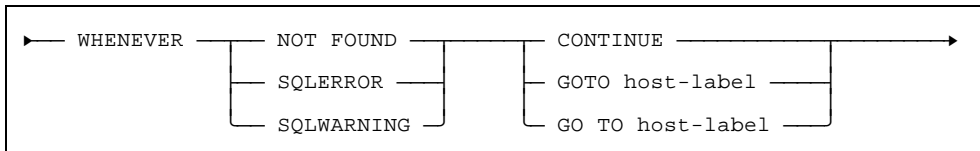
ALLOCATE CURSOR	DYNAMIC DELETE CURSOR
ALLOCATE DESCRIPTOR	DYNAMIC FETCH
ALTER DATABANK	DYNAMIC OPEN
ALTER DATABANK RESTORE	DYNAMIC UPDATE CURSOR
ALTER IDENT	ENTER
ALTER SHADOW	EXECUTE
ALTER TABLE	EXECUTE IMMEDIATE
CLOSE CURSOR	FETCH
COMMENT	GET DESCRIPTOR
COMMIT WORK	GET DIAGNOSTICS
CONNECT	GRANT
CREATE BACKUP	GRANT OBJECT PRIVILEGE
CREATE DATABANK	GRANT SYSTEM PRIVILEGE
CREATE DOMAIN	INSERT
CREATE IDENT	LEAVE
CREATE INCREMENTAL BACKUP	LEAVE RETAIN
CREATE INDEX	OPEN
CREATE SHADOW	PREPARE
CREATE SYNONYM	REVOKE
CREATE TABLE	REVOKE OBJECT PRIVILEGE
CREATE VIEW	REVOKE SYSTEM PRIVILEGE
DEALLOCATE DESCRIPTOR	ROLLBACK WORK
DEALLOCATE PREPARE	SELECT
DELETE CURSOR	SET CONNECTION
DELETE WHERE	SET DATABANK
DESCRIBE	SET DATABASE
DISCONNECT	SET DESCRIPTOR
DROP DATABANK	SET SHADOW
DROP DOMAIN	SET TRANSACTION CHANGES INVISIBLE
DROP IDENT	SET TRANSACTION CHANGES VISIBLE
DROP INDEX	SET TRANSACTION START EXPLICIT
DROP SHADOW	SET TRANSACTION START IMPLICIT
DROP SYNONYM	START TRANSACTION
DROP TABLE	UPDATE CURSOR
DROP VIEW	UPDATE WHERE
DYNAMIC CLOSE	UPDATE STATISTICS

8.3.2 Testing for run-time errors and exception conditions

The application program may test the outcome of a statement in one of two ways:

- by explicitly testing the content of the SQLSTATE variable
- by using the SQL statement **WHENEVER**, which tests the class of the SQLSTATE variable.

The **WHENEVER** statement has the general form:



The exception conditions detected are:

NOT FOUND row does not exist or result table is empty

SQLWARNING statement succeeded with warning status

SQLERROR statement failed

The action which may be taken is one of:

CONTINUE ignore the exception condition

GOTO label transfer program control to the line identified by the label.

The keyword **GOTO** may also be written as **GO TO**.

An application program may contain any number of **WHENEVER** statements, and the statements may be placed anywhere in the program. A separate **WHENEVER** statement must be issued for each situation (**NOT FOUND**, **SQLERROR** or **SQLWARNING**) which is to be tested. When an exception condition arises, action will be taken as specified in the **WHENEVER** statement most recently appeared in the code, for the respective condition.

WHENEVER statements are expanded by the preprocessor into explicit tests. These tests are placed after **every subsequent SQL statement in that program** until a new **WHENEVER** statement is issued for the same condition.

Two important consequences follow:

- **WHENEVER** statements are preprocessed strictly in the order in which they appear in the source code, regardless of execution order or conditional execution that the source code might imply. For instance, the **WHENEVER** statement in the following FORTRAN construction is expanded by the preprocessor, even though its 'execution' is never actually requested:

```

...
    GOTO 1025
EXEC SQL WHENEVER SQLERROR GOTO 1600
1025 CONTINUE
EXEC SQL DELETE FROM MYTABLE
...

```

- Mixing explicit tests and WHENEVER statements requires care. A 'WHENEVER *condition* CONTINUE' statement must be included before the SQL statement to which the WHENEVER statement applies, to cancel a 'hand-written' test of the SQLSTATE variable following the statement. As a general rule, it is advisable to use *either* hand-written tests *or* WHENEVER statements in a program module, and to avoid mixing them if possible.

A HOST LANGUAGE DEPENDENT ASPECTS

SQL statements may be embedded in any of the following host languages:

- C
- COBOL
- FORTRAN

Note that the machine specific User's Guide lists the ESQL preprocessors available for a specific environment.

This appendix describes features of embedded SQL which differ between the respective host languages. Note that it is not a complete description of the rules for writing embedded SQL programs. The programmer should use the main body of this manual as a guide to writing programs, and refer to this appendix for language-specific details.

The following topics are discussed for each language:

- SQL statement format: delimiters, margins, line continuation, comments, special characters.
- Restrictions.
- Host variables - declarations, SQL data type correspondence, value assignment rules.
- Preprocessor output format.
- Scope rules.

A.1 Embedded SQL in C programs

MIMER supports Embedded SQL for C following the ANSI standard.

A.1.1 SQL statement format

Statement delimiters

SQL statements are identified by the leading delimiter 'exec sql' and terminated by a semicolon (;).

Example:

```
exec sql DELETE FROM HOTEL;
```

Margins

Statements (including delimiters) may be written anywhere between positions 1 and 256 inclusive.

Line continuation

Line continuation rules for SQL statements are the same as those for ordinary C statements. Note however that the leading delimiter 'exec sql' may not be written over more than one line.

For a string constant, a white-space character (ASCII HEX-values 09 - 0D, or 20, i.e. <TAB>, <LF>, <VT>, <FF>, <CR> or <SP>), can be used to join two or more substrings. Each substring must be separately enclosed in delimiters.

Examples:

```
exec sql SELECT HOTELCODE, ROOMNO
        FROM   BOOK_GUEST
        WHERE  RESERVATION = :CUSTNO;
```

```
exec sql COMMENT ON TABLE ROOM_PRICES IS
        'Prices apply from date given'<LF>
        ' in column FROM_DATE';
```

Comments

Comments, enclosed between the markers /* and */, may be written anywhere within SQL statements where a white-space is permitted, except between the keywords 'exec' and 'sql' and within string constants. The comment may replace the white-space, for example

```
exec sql DELETE/* all rows */FROM HOTEL;
```

Special characters

The delimiters in SQL are apostrophes (') for string constants and quotation marks (") for delimited identifiers. This is contrary to the C string delimiter usage.

Keywords are separated by a white-space character.

A.1.2 Included code

If external source code modules containing SQL statements are to be included in the program, the non-standard INCLUDE statement must be used:

```
exec sql INCLUDE filename;
```

Files included in this way are physically integrated into the output from the preprocessor. Note that the file name must be enclosed in SQL string delimiters if it contains any non-alphanumerical characters.

A.1.3 SQLCA declaration

The SQLCA communication area is deprecated in version 7.3 of MIMER/SQL. If it is used, it must be declared with the non-standard INCLUDE SQLCA statement before any executable C or SQL statements are issued.

A.1.4 Host variables

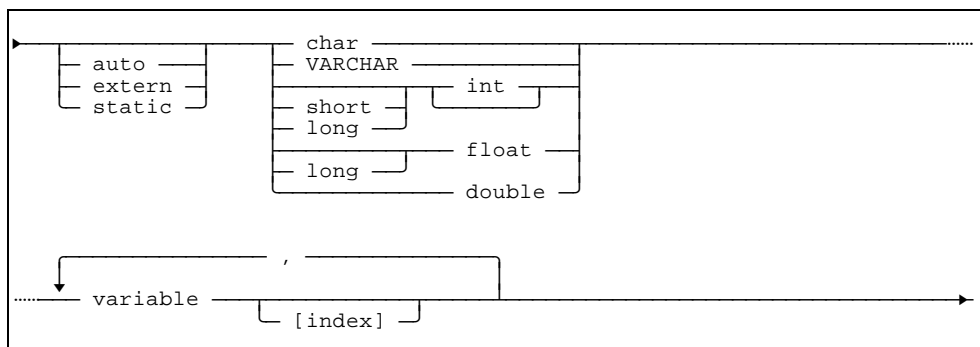
Declarations

Host variables used in SQL statements must be declared within the SQL DECLARE SECTION, delimited by the statements BEGIN DECLARE SECTION and END DECLARE SECTION.

Variables declared within the SQL DECLARE SECTION must conform to the following rules in order to be recognized by the SQL preprocessor:

- vector and array variables are not permitted with the exception of character arrays.
- host variables may be of AUTO, EXTERN or STATIC class, or parameters.
- character arrays are interpreted as null terminated strings.
- variables may not be explicitly initiated in the declaration.
- variable names are case significant.
- indicator variables must be declared as 2-byte integers (ordinarily 'short').

A syntax diagram showing the variable declarations recognized by the SQL preprocessor is given below:



Notes: In accordance with the syntax rules of C, keywords are case-sensitive and are given in the required case in the syntax diagram. This deviates from the general practice in MIMER documentation of using upper-case to denote keywords.

Index must be a number greater than 0 or the name of a numeric constant.

Only data type character can be indexed.

SQL data type correspondence

Valid host data types are listed below for each of the data types used in SQL statements.

SQL data type	C variable declaration
INTEGER	int short short int long long int
DECIMAL	float long float double
FLOAT	float long float double
CHARACTER	char VARCHAR
CHARACTER VARYING	char VARCHAR
DATETIME	char VARCHAR
INTERVAL	char VARCHAR

Note: Your C compiler may not support all of these possible declarations.

Value assignments

The general rules for conversion of values between compatible but different data types (see the MIMER/SQL Reference Manual, 'Basic SQL syntax rules') apply to the transfer of data between the database and host variables, with the data type correspondence as given in the table above.

When reading a character array host variable, the contents are conceptually padded with blanks from the first null byte to the length required. Note that the declared length is used as maximum length. When storing a value to a character array host variable, if the value is shorter than the variable length the value will be padded with blanks (and terminated with a null byte). See the MIMER/SQL Reference Manual, 'Basic SQL syntax rules', for a further discussion of character string assignments.

A.1.5 Preprocessor output format

Output from the C preprocessor retains SQL statements from the original source code as comments. Comments on the same line as SQL statements are retained as 'comments within comments', marked by the delimiters `/+ and +/`.

The preprocessed code is structured to reflect the structuring of the original source code.

A.1.6 Scope rules

Host variables follow the same scope rules as ordinary variables in C. SQL descriptor names, cursor names and statement names must be unique within the compilation unit. A compilation unit for C is the same as a file.

A.2 Embedded SQL in COBOL programs

MIMER supports Embedded SQL for COBOL following the COBOL-85 ANSI standard.

A.2.1 SQL statement format

Statement delimiters

SQL statements are identified by the leading delimiter 'EXEC SQL' and terminated by 'END-EXEC'.

SQL statements are treated exactly as ordinary COBOL statements with regard to the use of an ending period to mark the end of a COBOL sentence. Any valid COBOL punctuation may be placed after the END-EXEC terminator.

Examples:

```
EXEC SQL DELETE FROM HOTEL END-EXEC.
IF SQLSTATE NOT = "02000" THEN,
    EXEC SQL COMMIT END-EXEC,
ELSE
    EXEC SQL ROLLBACK END-EXEC.
```

Margins

Statements (including delimiters) may be written anywhere between positions 12 and 72 inclusive. Note in particular that if the leading delimiter 'EXEC SQL' starts before position 12, the statement will not be recognized as an SQL statement by the preprocessor.

Line continuation

Line continuation rules for SQL statements are the same as those for ordinary COBOL statements. Note however that the leading delimiter 'EXEC SQL' may not be written over more than one line.

If a string constant within an SQL statement is divided over several lines, the first non-blank character on the continuation line must be a string delimiter. There is no terminating string delimiter at the end of the line preceding the continuation line.

Example:

```
EXEC SQL SELECT HOTELCODE, ROOMNO
        FROM BOOK_GUEST
        WHERE RESERVATION = :CUST-NUMBER END-EXEC.
EXEC SQL COMMENT ON TABLE ROOM_PRICES IS
        'Prices apply from date given
-         ' in column FROM_DATE' END-EXEC.
```

An alternative way to break a character string constant over several lines, is to use a white-space character (ASCII HEX-values 09 - 0D, or 20, i.e. <TAB>, <LF>, <VT>, <FF>, <CR> or <SP>), to join two or more substrings. Each substring must be separately enclosed in delimiters.

```
EXEC SQL COMMENT ON TABLE ROOM_PRICES IS
        'Prices apply from date given'<LF>
        ' in column FROM_DATE' END-EXEC.
```

Comments

Comment lines, marked by an asterisk (*) in position 7, may be written within SQL statements. The whole line following a comment mark is treated as a comment.

Debugging lines and page eject lines (marked by D and / respectively in position 7) are treated as comments by the preprocessor.

Special characters

The delimiters in SQL are apostrophes (') for string constants and quotation marks (") for delimited identifiers. This is contrary to the default COBOL string delimiter usage.

If your COBOL compiler uses apostrophes (') as string delimiters, see the Users Guide for MIMER/SQL on the computer you are using to see how to indicate this to the preprocessor. String delimiters are described in more detail in Appendix B .

The default decimal point character in numerical constants in SQL statements is a period (.). If desired, a comma (,) may be used instead. See the Users Guide for MIMER/SQL on the computer you are using to see how to indicate this to the preprocessor.

Observe that the minus sign (-) is valid in variable names in COBOL. All arithmetic expressions using this operator should have at least one space separating the operands from the operator. Thus

A - B	means 'A minus B'
A-B	means 'variable called A-B'

A.2.2 Included code

Any code which is included into the program by the COBOL compiler is not recognized by the SQL preprocessor. If external source code modules containing SQL statements are to be included in the program, the non-standard SQL INCLUDE statement must be used:

```
EXEC SQL INCLUDE filename END-EXEC.
```

Files included in this way are physically integrated into the output from the preprocessor. Note that the file name must be enclosed in SQL string delimiters if it contains any non-alphanumerical characters.

A.2.3 Restrictions

The following restrictions apply specifically to COBOL:

- END-EXEC is a keyword reserved to SQL.
- COBOL figurative constants (such as ZERO and SPACE) may not be used as constants in SQL statements.

A.2.4 SQLCA declaration

The SQL communication area SQLCA is deprecated in version 7.3 of MIMER/SQL. If it is used, it must be declared in the WORKING STORAGE SECTION or LINKAGE SECTION of the DATA DIVISION, by issuing the non-standard INCLUDE SQLCA statement. The statement may be issued before or after host variable declarations:

```

...
DATA DIVISION.
WORKING STORAGE SECTION.
    ...
    EXEC SQL INCLUDE SQLCA END-EXEC.
    ...

```

A.2.5 Host variables

Declarations

Host variables used in SQL statements must be declared within the SQL DECLARE SECTION, delimited by the statements BEGIN DECLARE SECTION and END DECLARE SECTION.

Variables declared within the SQL DECLARE SECTION must conform to the following rules in order to be recognized by the SQL preprocessor:

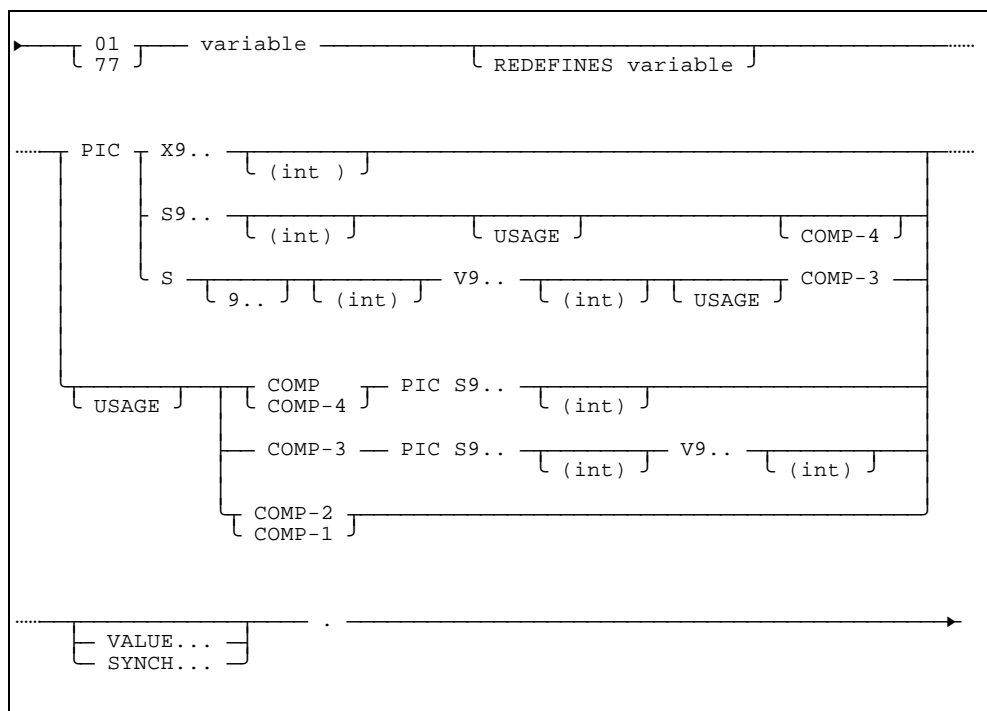
- variable names must begin with a letter. Within this restriction, any valid COBOL variable name may be used.
- host variable structures may not be used, except for varying-length character strings.
- the specifications JUSTIFIED, BLANK WITH ZERO, and OCCURS may not be used.
- the data type must be consistent with SQL data types as specified below.
- level number 01 or 77 should be used for all variable names which are used in SQL statements. Other levels may be used for program host variables, but they are not recognized by the preprocessor.
- FILLER entries are ignored for variables used in SQL statements.
- Indicator variables must be declared as 2-byte integers (PIC S9(n) COMP where $1 \leq n \leq 4$).
- Varying-length character structures (VARCHAR) are declared in the general form

```
01 name.
   49 len PIC S9(m) COMP.
   49 txt PIC X(n).
```

where `name` is the name of the VARCHAR structure
`len` is a half-word integer giving the current length of the character string
and `txt` contains the character string.

This is the only record structure recognized by the preprocessor.

A syntax diagram for COBOL variable declarations recognized by the SQL preprocessor is given below. Other declarations are ignored by the preprocessor:



Commas and semicolons may be used in accordance with standard COBOL practice. The following abbreviations are accepted

- | | | |
|-------|-----|-----------------------|
| PIC | for | PICTURE or PICTURE IS |
| USAGE | for | USAGE or USAGE IS |
| COMP | for | COMPUTATIONAL |
| SYNC | for | SYNCHRONIZED |

Note that the formulation PIC S9(n)9(m) is not accepted.

SQL data type correspondence

Valid host data types are listed below for each of the data types used in SQL statements.

Varying-length character string structures may be used in embedded SQL statements in COBOL programs. In assigning the value of such variables to columns, the current length of the string is used. The variable name used in SQL statements is the name of the structure (level 01 declaration), not of the character string element (level 49).

SQL data type	COBOL data declaration	Comments
INTEGER	01 name PIC S9(n) COMP.	$1 \leq n \leq 9$
DECIMAL	01 name PIC S9(n)V COMP-3. 01 name PIC S9(n)V9(m) COMP-3.	$1 \leq n \leq 15$ $1 \leq n, m \leq 15$
FLOAT	01 name COMP-2.	
CHARACTER	01 name PIC X(n). 01 name. 49 name PIC S9(m) COMP. 49 name PIC X(n).	$1 \leq n \leq 255$ $1 \leq m \leq 4$ $1 \leq n \leq 255$
CHARACTER VARYING	01 name PIC X(n). 01 name. 49 name PIC S9(m) COMP. 49 name PIC X(n).	$1 \leq n \leq 255$ $1 \leq m \leq 4$ $1 \leq n \leq 255$
DATETIME	01 name PIC X(n). 01 name. 49 name PIC S9(m) COMP. 49 name PIC X(n).	$1 \leq n \leq 255$ $1 \leq m \leq 4$ $1 \leq n \leq 255$
INTERVAL	01 name PIC X(n). 01 name. 49 name PIC S9(m) COMP. 49 name PIC X(n).	$1 \leq n \leq 255$ $1 \leq m \leq 4$ $1 \leq n \leq 255$

Value assignments

The general rules for conversion of values between compatible but different data types (see the MIMER/SQL Reference Manual, 'Basic syntax rules') apply to the transfer of data between the database and host variables, with the data type correspondence as given in the table above.

The first element in a varying-length character string structure is used to store the current length of the character string. When writing to the variable, the first element is updated with the current length of the variable. If the column value is longer than the variable, the value is truncated.

A.2.6 Preprocessor output format

Output from the COBOL preprocessor retains SQL statements from the original source code as comments. Comments within SQL statements are retained exactly as written. The output follows the ANSI standard for record format, and should be compiled with a COBOL compiler set to accept ANSI standard.

Debugging lines and page eject lines (using D and / respectively in position 7) remain unchanged after preprocessing.

The preprocessed code is structured to reflect the structuring of the original source code.

A.2.7 Scope rules

Host variables follows the same scope rules as ordinary variables in COBOL. SQL descriptor names, cursor names and statement names must be unique within the compilation unit. A compilation unit for COBOL is the same as a routine.

A.3 Embedded SQL in FORTRAN programs

MIMER supports Embedded SQL for FORTRAN following the FORTRAN-77 ANSI standard.

A.3.1 SQL statement format

Statement delimiters

SQL statements are identified by the leading delimiter 'EXEC SQL'. The end of an SQL statement is marked by the end of the line when the following line does not begin with a continuation character.

Example:

```
EXEC SQL DELETE FROM HOTEL
```

Margins

Source statements must be provided as fixed format, 80 byte records.

Statements (including delimiters) may be written anywhere between positions 7 and 72 inclusive. Note in particular that if the leading delimiter 'EXEC SQL' starts before position 7, the statement will not be recognized as an SQL statement by the preprocessor.

Line continuation

Line continuation rules for SQL statements are the same as those for ordinary FORTRAN statements. The continuation character is any character except space and 0 (zero) in position 6. Note however that the leading delimiter 'EXEC SQL' may not be written over more than one line. The FORTRAN limitation of a maximum of 19 continuation lines per statement does not apply within SQL statements.

For a string constant, a white-space character (ASCII HEX-values 09 - 0D, or 20, i.e. <TAB>, <LF>, <VT>, <FF>, <CR> or <SP>), can be used to join two or more substrings. Each substring must be separately enclosed in delimiters.

Examples:

```
EXEC SQL SELECT HOTELCODE, ROOMNO
+         FROM   BOOK_GUEST
+         WHERE  RESERVATION = :CUSTNO

EXEC SQL COMMENT ON TABLE ROOM_PRICES IS
+         'Prices apply from date given'<LF>
+         ' in column FROM_DATE'
```

Statement numbers

Any labelled SQL statement in the source code will generate a CONTINUE statement during preprocessing. Declarative SQL statements used before the first executable SQL statement should not be labelled.

Comments

Comment lines, marked by * or C in position 1, may be written within SQL statements. The whole line following a comment mark is treated as a comment, and the following line must either be another comment or follow the continuation rules given above.

Note that lines which are completely blank are not treated as comments by the FORTRAN preprocessor. The absence of a continuation character indicates the end of the previous statement, and a completely blank line may be used to structure comments in the output from the preprocessor. See Section A.4.7 for details.

Debugging lines (marked with a D in position 1) are treated as comments by the preprocessor.

A.3.2 Included code

Any code which is included into the program by the FORTRAN compiler (as directed by FORTRAN INCLUDE statements) is not recognized by the SQL preprocessor. If external source code modules containing SQL statements are to be included in the program, the non-standard SQL INCLUDE statement must be used:

```
EXEC SQL INCLUDE filename
```

Files included in this way are physically integrated into the output from the preprocessor. Note that the file name must be enclosed in SQL string delimiters if it contains any non-alphanumerical characters.

A.3.3 Restrictions

The following restriction applies specifically to FORTRAN:

- SQL statements may not be coded in logical IF-constructions.

A.3.4 SQLCA declaration

The SQL communication area SQLCA is deprecated in version 7.3 of MIMER/SQL. If it is used, it must be declared with the INCLUDE SQLCA statement before any executable FORTRAN or SQL statement is issued.

The preprocessor will insert DATA statements directly following the SQLCA declaration. Note that some FORTRAN compilers may not accept declarations which follow the first DATA statement. The INCLUDE SQLCA statement should therefore be placed after all host declarations and before DATA statements if the program is to follow strict FORTRAN standards.

A.3.5 Host variables

Declarations

Host variables used in SQL statements must be declared within the SQL DECLARE SECTION, delimited by the statements BEGIN DECLARE SECTION and END DECLARE SECTION.

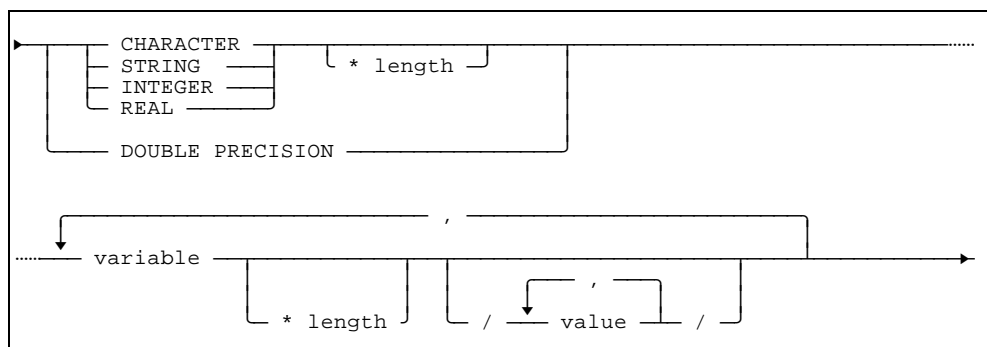
Variables declared within the SQL DECLARE SECTION must conform to the following rules in order to be recognized by the SQL preprocessor:

- any valid FORTRAN variable name may be used.
- variables must be scalar variables (i.e. they may not be elements of vectors or arrays).
- implicit declaration by means of the IMPLICIT statement or default typing may not be used.
- FORTRAN COMPLEX variables may not be used.
- character variables must be declared with a fixed constant length. Expressions and variable length declarations (such as CHARACTER*(*)) may not be used.
- indicator variables must be declared as half-word integers (INTEGER*2).

Character variables may be declared as standard FORTRAN-77 CHARACTER, but such variables may only be used in FORTRAN-77 character handling routines, not in MIMER library routines.

Some machine-specific implementations may allow variables declared as CHARACTER to be passed to routines requiring Hollerith type parameters (used in FORTRAN-66). Programs written with such techniques are however not necessarily portable between different machines.

A syntax diagram showing the variable declarations recognized by the SQL preprocessor is given below:



Note the non-standard data type STRING, used to force Hollerith-type declarations from the preprocessor (see below).

The data type declaration must be separated from the variable name by at least one space (which is not required in FORTRAN declarations outside the SQL DECLARE SECTION). Thus the declaration

```
INTEGER*2A
```

is not recognized. The required formulation is

```
INTEGER*2 A
```

Lists of variables following a single default data type declaration are accepted. Any declarations in a list which are not valid in SQL contexts are ignored by the preprocessor. Thus the following statement declares variables A and D as INTEGER*4 and B as INTEGER*2 for use in SQL statements, while the array C is ignored:

```
INTEGER*4 A, B*2, C(10), D
```

SQL data type correspondence

Valid host data types are listed below for each of the data types used in SQL statements.

SQL data type	FORTTRAN data declaration
INTEGER	INTEGER*2 INTEGER*4
DECIMAL	REAL*4 REAL*8 DOUBLE PRECISION
FLOAT	REAL*4 REAL*8 DOUBLE PRECISION
CHARACTER	CHARACTER*n STRING*n
CHARACTER VARYING	CHARACTER*n STRING*n
DATETIME	CHARACTER*n STRING*n
INTERVAL	CHARACTER*n STRING*n

The following additional points should be noted:

- Character variables to be used with routines from the MIMER Runtime Libraries (i.e. LR-, FM-, and RG-routines) must be declared as the special STRING type. The FORTRAN preprocessor converts this declaration into a Hollerith string declaration (an integer array). Such STRING variables may not be used with FORTRAN-77 character handling routines. The MIMER LR-routines provide character handling facilities which may be used in place of FORTRAN-77 routines. Since Hollerith strings are not recognized by the preprocessor, string declarations cannot contain any value specifications. However, string variables can, like all variables, be initiated in separate data statements where each word in the generated integer array is initiated by a Hollerith literal.
- FORTRAN does not support DECIMAL data types. A string of digits including a decimal point is interpreted as a REAL constant in FORTRAN. Exponential notation should always be used to specify floating point values in SQL statements.
- DOUBLE PRECISION constants may be written with a D as the exponent marker in FORTRAN (e.g. 1.23D+02). The only permissible exponent marker within SQL statements is E (e.g. 1.23E+02).

Value assignments

The general rules for conversion of values between compatible but different data types (see the MIMER/SQL Reference Manual, 'Basic syntax rules') apply to the transfer of data between the database and host variables, with the data type correspondence as given in the table above.

A.3.6 Preprocessor output format

Output from the FORTRAN preprocessor retains SQL statements from the original source code as comments. The output follows the ANSI standard for record format, and should be compiled with a FORTRAN compiler set to accept ANSI standard. Comments within SQL statements are retained exactly as written.

Completely blank lines between SQL statements and following comments cause the preprocessor to write the comments after the generated SQL call. Otherwise comments immediately following SQL statements are output before the generated call. Debugging lines (using D in position 1) remain unchanged after preprocessing.

The preprocessed code is structured to reflect the structuring of the original source code.

A.3.7 Scope rules

Host variables follows the same scope rules as ordinary variables in FORTRAN. SQL descriptor names, cursor names and statement names must be unique within the compilation unit. A compilation unit for FORTRAN is the same as a routine.

B PREPROCESSING APPLICATION PROGRAM SOURCE CODE

This appendix describes aspects of preprocessing source code for embedded application programs common to all MIMER/SQL implementations. Details of how the preprocessors are run, input and output file name standards, and syntax for the preprocessor switches are machine-dependent, and are described in the MIMER User's Guide for the particular computer in question.

B.1 Input and output files

The function of the preprocessor is to convert SQL statements embedded in host language source code into calls to the SQL compiler. In all host languages, embedded SQL statements are identified by the leading keywords EXEC SQL, which must be written on the same line (see Appendix A). There is one preprocessor for each host language supported.

The input to the preprocessor is thus a source code file containing host language statements and embedded SQL statements. The output from the preprocessor is a source code file in the same host language, with the embedded SQL statements converted to data assignment statements and subroutine calls. The original embedded SQL statements are retained as comments in the output file, to help in understanding the program if a source code debugger is used.

The output file from the preprocessor is used as input to the host language compiler to produce object code.

Note: The programmer should never modify the code produced by the preprocessor. If changes are to be made to the program, the original embedded source code should be modified and preprocessed again. Sysdeo Mimer cannot accept responsibility for the consequences of manual modification of preprocessor output.

B.2 Preprocessor options

For certain host languages, options (switches) may be set for the preprocessor to determine the way the embedded source code is handled. The switch functions are summarized for each host language below. Refer to the machine-specific User's Guide for the switch syntax.

B.2.1 C

There are no switches for the C preprocessor.

B.2.2 COBOL

String delimiters

Some COBOL compilers do not follow the ANSI standard with regards to the string delimiter. If required, you may reverse the usage of apostrophes and quotation marks as string delimiters in COBOL statements, to conform to the usage chosen for your COBOL compiler. The default usage is as follows:

Standard:	SQL standard	COBOL ANSI standard
String delimiter:	'	"

Note that the COBOL ANSI standard usage is reversed in relation to the SQL standard.

The preprocessor ensures that the correct delimiters are supplied to the COBOL compiler, provided that you use the switch correctly.

Decimal point character

The normal decimal point character in SQL statements is a period (.). The COBOL preprocessor may be set to accept a comma (,) as the decimal point character in SQL statements, to conform to the standard COBOL usage.

Note that the switch affects only decimal point characters in SQL statements. Decimal point characters in COBOL statements are never changed by the preprocessor.

B.2.3 FORTRAN

SAVE statements

The FORTRAN preprocessor normally generates FORTRAN-77 SAVE statements for local variables. This function may be turned off with a preprocessor switch, so that preprocessed code may be submitted to FORTRAN compilers which do not recognize SAVE statements.

C RETURN CODES

MIMER/SQL returns two kinds of return codes to an application. The `SQLSTATE` variable returns a standardized, general error code, which gives a rough description of the status for the most recently executed SQL statement. If `GET DIAGNOSTICS` is called to get an error message, a more descriptive return code is retrieved along with the error message. This return code is an internal MIMER return code, and it gives a more precise description of the result for the most recently executed SQL statement.

C.1 SQLSTATE return codes

SQLSTATE contains a 5-character long return code string that indicates the status of an SQL statement. These return codes are standardized following the established standards. Observe that not all standardized SQLSTATE return codes are used by MIMER/SQL.

The SQLSTATE values consists of two fields. The class, which is the first two characters of the string, and the subclass, which is the terminating three characters of the string.

List of SQLSTATE values:

Class	Subclass	Meaning
00	000	Success
01	000	Success with warning
01	002	- Disconnect error
01	003	- Null value eliminated in set function
01	004	- String data, right truncation
01	005	- Insufficient item descriptor areas
01	006	- Privilege not revoked
01	007	- Privilege not granted
02	000	No data
07	000	Dynamic SQL error
07	001	- <i>using-clause</i> does not match dynamic variables
07	002	- <i>using-clause</i> does not match target specification
07	003	- Cursor specification cannot be executed
07	008	- Invalid descriptor count
07	009	- Invalid descriptor index
08	000	Connection exception
08	001	- Client unable to establish connection
08	002	- Connection name in use
08	003	- Connection does not exist
08	004	- Server rejected the connection
08	006	- Connection failure
0A	000	Feature not supported
21	000	Cardinality violation

Class	Subclass	Meaning
22	000	Data exception
22	001	- String data, right truncation
22	002	- Null value, no indicator variable
22	003	- Numeric value out of range
22	005	- Error in assignment
22	006	- Invalid interval format
22	007	- Invalid datetime format
22	008	- Datetime field overflow
22	011	- Substring error
22	012	- Division by zero
22	015	- Interval field overflow
22	018	- Invalid character value for CAST
22	019	- Invalid escape character
22	023	- Invalid parameter value
22	025	- Invalid escape sequence
22	027	- Trim error
23	000	Integrity constraint violation
24	000	Invalid cursor state
25	000	Invalid transaction state
26	000	Invalid SQL statement identifier
28	000	Invalid authorization specification
2E	000	Invalid connection name
33	000	Invalid SQL descriptor name
34	000	Invalid cursor name
35	000	Invalid exception number
37	000	Syntax error or access violation (in PREPARE or EXECUTE IMMEDIATE)
3C	000	Ambiguous cursor name
40	000	Transaction rollback
40	001	- Serialization failure
40	003	- Statement completion unknown
42	000	Syntax error or access violation
44	000	WITH CHECK OPTION violation
S1	000	General error (ODBC)
S1	001	- Memory allocation failure

C.2 Internal MIMER return codes

Here the internal MIMER return code values are listed together with the associated text message. These return codes are retrieved together with the error message when GET DIAGNOSTICS is called. The symbol <%> in a text message indicates the location of an identifier inserted at run-time when the DBERM4 routine is used to display the message text. The codes are grouped according to function as follows:

Code numbers	Functional group
> 0	Warnings, messages
= 0	Success
-100 to -999	ODBC error codes
-10000 to -10999	Data-dependent errors
-11000 to -11999	Limits exceeded
-12000 to -12999	SQL statement errors
-13000 to -13999	Not currently used
-14000 to -14999	Program-dependent errors
-15000 to -15999	Not currently used
-16000 to -16999	Databank and table errors
-17000 to -17999	Not currently used
-18000 to -18999	Miscellaneous errors
-19000 to -19999	Internal errors

Corrective action is given in general terms for each group of codes. If you cannot correct the error, first try to locate the cause of the error in your source code, and then contact your MIMER representative giving a full description of the error code and the application program context in which it arose.

C.3 Warnings and messages

No corrective action is normally required for internal MIMER return code values greater than zero.

90	Login failure
91	Soft enter performed
92	No cursor state was saved on stack
93	NULL values were eliminated from the argument of a function
94	Message text not found
100	No rows found
101	No data - Item number is greater than the value of count

C.4 ODBC errors

These errors arise when ODBC calls to MIMER fails for some reason.

-100	Illegal sequence
-101	Out of memory
-102	Option out of range
-103	Function not supported
-104	Connection not open
-105	Connection in use
-106	Invalid argument value
-107	Invalid transaction operation code
-108	Internal network buffer overflow
-109	Invalid C data type
-110	Invalid SQL data type
-111	Bad address
-112	Function already active
-113	Operation cancelled
-114	Wrong number of parameters
-115	Use ODBC function SQLTransact to commit or rollback transaction
-116	Missing naming descriptor
-117	Invalid transaction state
-118	Unknown statement type
-119	Unknown internal data type
-120	Extype corrupt
-121	Invalid buffer length
-122	String data truncated
-123	Numeric data truncated
-124	Significant digits lost
-125	Invalid numeric value
-126	Bad parameter passed to numeric package
-127	Invalid column number
-128	Database name mandatory
-129	Connect dialog failed
-130	Data truncated
-131	Invalid connection string attribute
-132	Invalid cursor state
-133	Invalid parameter number
-134	Descriptor type out of range
-135	Invalid type passed to DICOA3
-136	Function type out of range
-137	Invalid cursor name
-138	Duplicate cursor name
-139	Cursor hash table corrupt
-140	ODBC database control block chain corrupt
-141	Option type out of range

- 142 Option value not supported
- 143 Option not supported
- 144 Invalid row or keyset size
- 145 Invalid concurrency option
- 146 Invalid fetch type
- 147 Not a scrollable cursor
- 148 Row position out of range
- 149 Only one SQLPutData for fixed length parameter
- 150 SQLPutData and SQLGetData do not support bulk operations
- 151 Driver not capable
- 152 Table type out of range
- 153 Invalid string length
- 154 Data type out of range
- 155 Syntax error found in escape clause
- 156 DDO buffer overflow
- 157 Uniqueness option type out of range
- 158 Accuracy option type out of range
- 159 Column type out of range
- 160 Scope type out of range
- 161 Nullable type out of range
- 162 Internal type mismatch
- 163 Conversion between data types not supported
- 164 Invalid date, time, or timestamp
- 165 Restricted data type attribute violation
- 166 Date, time, or timestamp data truncated
- 167 Database has not been configured. Run Configure MIMER 7.1
- 168 Translated native SQL string was truncated
- 169 ODBC extension DATE, TIME or TIMESTAMP is not supported
- 170 ODBC extension OUTER JOIN is not supported
- 171 ODBC extension for procedure invocation is not supported
- 172 Unrecognized first word in escape clause, expected 'FN', 'OJ', 'D', 'T' or 'TS'
- 173 MIMER 7.1 servers do not support the used scalar function
- 174 Unrecognized scalar function found in escape clause
- 175 Argument missing in scalar function
- 176 Too many arguments in scalar function
- 177 Syntax error, incomplete escape clause
- 178 Syntax error, unmatched apostrophe in string literal
- 179 Syntax error, unmatched quote in delimited identifier
- 180 Invalid data type specified in scalar function CONVERT
- 181 Information type out of range
- 182 Parameter type may only be used with procedures
- 183 Parameter type out of range
- 184 Update and delete where current fully supported (not simulated)
- 185 Option value changed

- 186 Static scrollable cursor used instead of keyset or dynamic cursor
- 187 Error in row, please check next error code
- 188 Cancel treated as FreeStatement/CLOSE
- 189 Attempt to fetch before the result set returned the first rowset
- 180 Invalid cursor position
- 191 Unknown first parameter in scalar function TIMESTAMPADD
- 192 Unknown first parameter in scalar function TIMESTAMPDIFF
- 193 Bad parameter to datetime package

C.5 Data-dependent errors

These errors arise when an SQL statement cannot be executed correctly because of the data content of variables, expressions, and so on in the statement. The appropriate corrective action is determined by the nature of the error and the specific context in the application program.

- 10001 Transaction aborted due to conflict with other transaction
- 10101 INSERT operation invalid because the resulting table will contain a primary key duplicate
- 10102 Domain constraint violation
- 10103 Table or column constraint violation
- 10104 View constraint violation
- 10105 INSERT or UPDATE operation invalid because the referencing table <%> does not satisfy referential constraints
- 10106 UPDATE or DELETE operation invalid because the referenced table <%> does not satisfy referential constraints
- 10107 The result of a subquery in a basic predicate is more than one value
- 10108 Result of a SELECT INTO statement is a table of more than one row
- 10109 Type constraint violation
- 10110 Unique constraint violation
- 10199 Host variable type packed decimal is not supported
Please refer to users guide for further information
- 10200 Reserved numeric operand found during data type conversion
- 10201 Length error or incorrect value found during data type conversion
- 10202 Division by zero attempted
- 10203 Negative overflow occurred during data type conversion
- 10204 Positive overflow occurred during data type conversion
- 10205 Loss of significance occurred during data type conversion
- 10207 Undefined value found during data type conversion
- 10211 Undefined value found during data type conversion
- 10212 Overflow occurred during data type conversion
- 10221 The NULL value cannot be assigned to a host variable because no indicator variable is specified
- 10222 NULL not allowed for item descriptor area
- 10301 Loss of significance occurred in arithmetic operation <%>
- 10302 Positive overflow occurred in arithmetic operation <%>

- 10303 Negative overflow occurred in arithmetic operation <%>
- 10304 Division by zero attempted
- 10305 Bad parameter encountered in arithmetic operation <%>
- 10306 Invalid input for numeric function
- 10310 Invalid character value for CAST
- 10311 String data truncated
- 10312 Numeric value out of range
- 10313 Illegal (negative) substring length
- 10321 Datetime loss of significance
- 10322 Datetime positive overflow
- 10323 Datetime negative overflow
- 10325 Bad parameter encountered in datetime arithmetic operation
- 10326 Datetime illegal operand
- 10327 Invalid datetime value
- 10328 Datetime subtype mismatch
- 10329 Invalid datetime format
- 10331 Interval loss of significance
- 10332 Interval positive overflow
- 10333 Interval negative overflow
- 10335 Bad parameter encountered in interval arithmetic operation
- 10336 Interval illegal operand
- 10337 Invalid interval value
- 10338 Interval subtype mismatch
- 10339 Invalid interval format
- 10601 Invalid value for field of item descriptor area

C.6 Limits exceeded

These errors arise when internal limits in the MIMER system are exceeded. Some of the limitations are determined by installation-specific parameters, while others are fixed by MIMER. In general, errors of this nature require either re-installation of the system with extended limitations or modification of the application program to reduce the system demands. Contact your MIMER representative if you have difficulty avoiding errors of this nature.

- 11001 Dynamic storage area exhausted in host level interface (DYNDE3)
- 11002 Internal DB dynamic storage area exhausted (DSDYN2)
- 11011 Internal storage for like pattern exhausted
- 11012 Transaction list exhausted
- 11013 Too many databanks referenced in statement (max 30)
- 11014 Too many databanks active in transaction
- 11100 Internal limit exceeded : query stack
- 11101 Internal limit exceeded : scan stack
- 11102 Internal limit exceeded : generation stack
- 11103 Internal limit exceeded : table descriptor list
- 11104 Internal limit exceeded : data storage

- 11105 Internal limit exceeded : patch table
- 11106 Internal limit exceeded : label table
- 11107 Internal limit exceeded : traversal stack
- 11108 Internal limit exceeded : sco list
- 11109 Internal limit exceeded : boolean stack
- 11110 Internal limit exceeded : index list
- 11111 Internal limit exceeded : semantic stack
- 11112 Internal limit exceeded : working storage (DSDYN3)
- 11113 Internal limit exceeded : query too complex
- 11114 Required temporary table row length is <%>, only <%> is possible
- 11115 Internal limit exceeded : restriction group pool
- 11116 Escape clause parser stack exhausted
- 11117 Internal limit exceeded : join stack
- 11118 Internal limit exceeded : scan queue

C.7 SQL statement errors

These errors arise from syntactic or semantic errors in SQL statements. In general, syntactic errors in embedded SQL programs are detected by the preprocessor, so errors cannot arise at run-time. Dynamically submitted SQL statements are however parsed at run-time, and the syntax error codes are returned after attempting PREPARE for a syntactically incorrect source statement.

Semantic errors can arise at run-time from both dynamic and static SQL statements.

- 12001 Too many errors, error collection terminated
- 12101 Syntax error, <%> assumed missing
- 12102 Syntax error, <%> ignored
- 12103 Syntax error, <%> assumed to mean <%>
- 12104 Invalid construction
- 12106 Internal parser error, analysis aborted
- 12107 Syntax analysis resumed here
- 12108 Multiple statements not allowed
- 12120 Table name too long
- 12121 String literal too long
- 12122 Numeric literal too long
- 12123 Invalid password string
- 12124 Invalid hexadecimal literal
- 12125 Reserved word may not be used as an identifier
- 12126 Invalid name
- 12128 Result of concatenation too long
- 12129 Table definition does not include any column specification
- 12131 Table definition includes more than one primary key specification
- 12132 Only one column allowed in column list
- 12133 Max precision for CURRENT TIME & CURRENT TIMESTAMP is 6

- 12152 <%> not allowed in EXECUTE mode
- 12154 User name too long
- 12156 Column name too long
- 12157 Synonym name too long
- 12158 Correlation name too long
- 12159 Cursor name too long
- 12160 Databank name too long
- 12161 Shadow name too long
- 12162 Host Variable or Parameter Marker name too long
- 12163 File name too long
- 12164 Label name too long
- 12165 Index name too long
- 12166 Object name too long
- 12167 View name too long
- 12168 Domain name too long
- 12169 Too many identifier names given
- 12170 SQL1 construction, <%>, not yet implemented
- 12171 <%> key must be specified as NOT NULL
- 12172 UNION not allowed in combination with scrollable cursor
- 12173 Invalid escape sequence
- 12174 Syntax error in escape clause, expecting comma before PRODUCT or CONFORMANCE specification
- 12175 Syntax error in escape clause, invalid CONFORMANCE specification
- 12176 Syntax error in escape clause, invalid YEAR specification
- 12177 Syntax error in escape clause, invalid PRODUCT specification
- 12178 Syntax error in escape clause, invalid VENDOR specification
- 12179 Syntax error in escape clause, expecting VENDOR or YEAR after '--(*)'
- 12180 Syntax error, unexpected token '*)--'
- 12181 Syntax error in escape clause, terminating '*)--' missing
- 12182 Syntax error in escape clause
- 12200 Table <%> not found, table does not exist or no access privilege
- 12201 Table reference <%> is ambiguous
- 12202 <%> is not a column of an inserted table, updated table or any table identified in a FROM clause
- 12203 <%> is neither an object table of an INSERT, UPDATE or DELETE statement, nor specified in a FROM clause
- 12204 Column reference <%> ambiguous
- 12205 Column <%> not referenced in GROUP BY clause
- 12206 <%> clause not allowed in a subquery of a basic predicate
- 12207 DISTINCT specified more than once in a subselect
- 12208 SELECT clause of a subquery specifies more than one column
- 12209 Column <%> identified in HAVING clause but not included in GROUP BY clause
- 12210 Operand of set function includes a set function
- 12211 Operand of set function does not include a column name

- 12212 Operand of set function includes a correlated reference specified in an expression
- 12213 Set function not specified in a SELECT clause or HAVING clause
- 12214 Invalid operand type, expected type is <%>
- 12215 Operand not of <%> type
- 12216 Operands are not comparable
- 12217 Set function containing DISTINCT may not be specified within an expression
- 12220 Expression must be a column
- 12221 SELECT clause contains both column expressions and set function expressions
- 12223 ORDER BY clause invalid because it includes a column name that is not part of the result table
- 12224 ORDER BY clause invalid because it includes an integer which does not identify a column of the result table
- 12225 The ORDER BY clause invalid because it includes an ambiguous column reference
- 12226 Set function argument not bound in HAVING context
- 12227 Duplicate column reference in FOR UPDATE OF clause
- 12228 <%> is identified in a FROM clause and is also referencing the object table or view of an INSERT, UPDATE or DELETE statement
- 12229 Column <%> cannot contain NULL values
- 12230 Invalid numeric literal
- 12231 Update or insert value is NULL, but the object column <%> cannot contain NULL values
- 12232 Insert value must be a constant expression or NULL
- 12233 The number of insert values is not the same as the number of object columns
- 12234 Update or insert value not compatible with the data type of the object column <%>
- 12235 <%> is the object table of an INSERT statement and may therefore not be referenced in a subselect
- 12236 Column name <%> does not identify a unique column of the result table
- 12238 Column <%> cannot be updated because it is either included in the primary key or derived from a set function or expression
- 12239 The use of NULL in a SELECT clause is only allowed in a UNION
- 12240 Statement contains too many table references
- 12242 The corresponding columns of the operands of a UNION do not have compatible column descriptions
- 12243 Result table contains a column for which the type cannot be determined
- 12244 Operands of a UNION do not have the same number of columns
- 12245 FOR UPDATE clause may not be specified because the result table cannot be modified
- 12246 Column <%> in the FOR UPDATE clause is not part of the identified table or view

- 12247 Numeric literal or calculated numeric value outside range of object column
- 12250 A host variable or parameter marker is not allowed in a view definition
- 12251 CREATE VIEW statement must include a column list because the SELECT clause contains an expression
- 12252 CREATE VIEW statement must include a column list because the SELECT clause contains duplicate column names
- 12253 The number of columns specified for the view is not the same as specified by the SELECT clause
- 12254 WITH CHECK OPTION cannot be used for the specified view because it cannot be modified
- 12255 A join may not include a grouped view
- 12256 The FROM list of a grouped query may not name a grouped view
- 12257 A view containing GROUP BY or HAVING may not be identified in a FROM clause of a subquery of a basic predicate
- 12258 <%> operation not permitted because the view cannot be modified
- 12259 <%> operation not permitted because the joined table cannot be modified
- 12260 A view column contained in a WHERE clause may not be derived from a set function or be a grouping column
- 12261 INSERT statement not permitted because the object column <%> is derived from an expression
- 12262 The type of the parameter marker cannot be determined
- 12263 Parameter markers and host variables not allowed in EXECUTE IMMEDIATE environment
- 12264 Parameter markers may not be specified in SELECT clause
- 12265 Literal or computed value overflow
- 12266 Decimal divide operation invalid because the result would have a negative scale
- 12267 Duplicate column reference in INSERT column list
- 12268 Duplicate column reference in UPDATE set clauses
- 12269 Duplicate column reference in View Parameter
- 12270 <%> does not have <%> privilege on object <%>
- 12271 Duplicate column reference in GROUP BY clause
- 12273 The types of the results of a CASE expression are not type compatible
- 12274 The type of the CASE expression result cannot be determined
- 12275 At least one result in a CASE expression must be non-null
- 12276 A view is not allowed in a qualified JOIN
- 12277 Invalid CAST datatype specification
- 12278 Invalid EXTRACT field specification
- 12280 Invalid datetime literal
- 12281 Invalid interval literal
- 12282 Invalid interval qualifier
- 12283 Invalid use of interval qualifier
- 12500 Databank named <%> already exists

- 12501 Table <%> does not exist
- 12502 <%> does not have <%> privilege
- 12503 <%> does not have <%> privilege on object <%>
- 12504 Statement not allowed within transaction
- 12505 <%> is not a USER or PROGRAM ident
- 12506 No privilege
- 12509 An ident cannot REVOKE a privilege from itself
- 12510 An ident cannot GRANT a privilege to itself
- 12511 Duplicate column specification
- 12512 Invalid type description
- 12514 Primary key not valid
- 12515 Maximum row length exceeded by index table
- 12516 Qualified column name required
- 12517 Object <%> does not exist
- 12518 Circular grant of membership between groups not permitted
- 12519 GRANT OPTION missing for privilege
- 12520 An ident cannot GRANT a privilege to itself
- 12521 Same table privilege used repeatedly
- 12522 No such privilege held
- 12523 Databank <%> does not exist
- 12526 Default value for NOT NULL column <%> missing
- 12529 Column <%> may not have a NOT NULL constraint
- 12530 Operand not of type <%>
- 12531 Operands not comparable
- 12533 Literal or computed value overflow
- 12534 Invalid numeric literal
- 12535 Invalid identifier, keyword VALUE expected
- 12536 Name <%> in PRIMARY KEY clause not recognized as a column name of current table definition
- 12537 <%> must be unqualified
- 12538 Default value not compatible with domain definition
- 12539 Host variable construction illegal in this context
- 12540 <%> is not a column of the specified table(s)
- 12541 No valid change
- 12542 Default value is outside the range specified by domain definition
- 12543 UPDATE privilege on individual columns may not be granted with GRANT OPTION
- 12544 Too many columns specified in <%> statement
- 12545 Primary key column <%> may not be updated
- 12546 Column <%> is not type compatible with the corresponding column of the referenced table
- 12547 Number of columns specified in the foreign key is not the same as the number of columns in the primary key of the referenced table
- 12548 <%> privilege for specific column may not be revoked
- 12549 Databank option may not be changed to NULL since <%> contains tables with foreign key, reference or unique constraints

- 12550 Table <%> includes a foreign key and may therefore not be created in a databank with NULL option
- 12551 Table <%> is in a databank with NULL option and may therefore not be used as reference table
- 12552 Creator name <%> must be equal to the name of the current user
- 12553 Explicit grant membership on PUBLIC is not permitted
- 12554 PUBLIC cannot be member of another group
- 12556 <%> cannot be shadowed because it is a NULL databank
- 12557 Shadow named <%> already exists
- 12558 Ident named <%> already exists
- 12559 Index named <%> already exists
- 12560 Table, View, Synonym or Domain named <%> already exists
- 12561 Domain <%> does not exist
- 12563 Shadow <%> does not exist
- 12564 Ident <%> does not exist
- 12565 Maximum row length exceeded by foreign key table
- 12566 Maximum row length exceeded by base table
- 12568 EXISTS construction illegal in this context
- 12569 ALL or ANY construction illegal in this context
- 12570 Set function construction illegal in this context
- 12571 Subquery construction illegal in this context
- 12572 Too many foreign key columns given
- 12573 Name <%> in CHECK clause not recognized as a column name of current table definition
- 12574 Name <%> in column constraint not recognized as a current column name
- 12575 Column <%> may not have a column CHECK constraint
- 12576 Table <%> includes a unique index and may therefore not be created in a databank with NULL option
- 12577 Default value not compatible with column specification
- 12578 Column <%> does not exist in referenced table
- 12579 No such key in referenced table
- 12580 Table <%> includes an alternate key and may therefore not be created in a databank with NULL option
- 12581 Too many alternate key columns given
- 12582 Alternate key equivalent to primary key
- 12583 Alternate key equivalent to previously given alternate key
- 12584 Number of columns specified in the foreign key is not the same as the number of columns in the specified alternate key of the referenced table
- 12585 Name <%> in FOREIGN KEY clause not recognized as a column name of current table definition
- 12586 Column <%> in ALTER TABLE statement may not have a DEFAULT value
- 12587 Column in ALTER TABLE statement may not be specified as PRIMARY KEY

- 12588 Column in ALTER TABLE statement may not be specified as UNIQUE
- 12589 Column in ALTER TABLE statement may not have a REFERENCES part
- 12590 Table contains too many columns
- 12591 Cannot create unique index
- 12592 Depending objects exist when RESTRICT specified
- 12593 Column <%> does not exist
- 12594 Column <%> cannot be dropped as it is the only column in table
- 12595 Column <%> cannot be dropped, restrictions exist
- 12596 Default value for column <%> does not exist
- 12597 Column <%> cannot be dropped because it is part of the primary key

- 12601 Statement does not support backup of <%>
- 12602 The same file name is given for backup and incremental backup
- 12603 Database is already OFFLINE
- 12604 Database is already ONLINE
- 12605 Cannot RESET LOG, because database is ONLINE
- 12606 Databank <%> is already OFFLINE
- 12607 Databank <%> is already ONLINE
- 12608 Cannot RESET LOG, because databank <%> is ONLINE
- 12609 Shadow <%> is already OFFLINE
- 12610 Shadow <%> is already ONLINE
- 12611 Cannot RESET LOG, because shadow <%> is ONLINE
- 12612 Shadow <%> is already specified
- 12613 Cannot set more than one shadow OFFLINE for databank having shadow <%>
- 12614 Statistics cannot be updated for <%> because it is a view

C.8 Program-dependent errors

These errors arise as a result of incorrect program construction. In general, corrective action requires revision of the program source code.

- 14001 Invalid sequence of SQL statements
- 14002 SQL statement invalid because the user is not connected
- 14003 CONNECT statement invalid because the user is already connected
- 14004 System already closed down
- 14005 Cannot perform DISCONNECT in a transaction
- 14011 Transaction already started
- 14012 Transaction handling required
- 14013 No transaction started
- 14021 Cannot perform ENTER or LEAVE in a transaction
- 14022 Cannot perform ENTER operation because program level is already active
- 14023 No program level entered, cannot perform leave operation

- 14031 DESCRIBE statement does not identify a prepared statement
- 14032 EXECUTE statement does not identify a prepared statement
- 14033 PREPARE statement identifies a SELECT statement of an opened cursor
- 14034 The cursor is not in a prepared state
- 14035 The cursor identified in a FETCH or CLOSE statement is not open
- 14036 The cursor cannot be used because its statement name does not identify a prepared SELECT statement
- 14037 The cursor identified in the UPDATE or DELETE statement is not open
- 14038 UPDATE or DELETE CURRENT statement not allowed for a cursor of a prepared SELECT statement
- 14039 Cursor is not scrollable
- 14041 The cursor identified in the UPDATE statement is not positioned on a row
- 14042 The cursor identified in the DELETE statement is not positioned on a row
- 14101 Invalid statement identifier
- 14201 Compilation did not yield an executable program
- 14202 The cursor identified in an OPEN statement is already open, but not declared as REOPENABLE
- 14203 Statement position cannot be saved when temporary tables are used in the query
- 14301 SQLDA contains an invalid data address or indicator variable address
- 14302 Invalid address of username or password
- 14303 Invalid address
- 14311 Illegal statement length given for SQL statement
- 14312 Input character string too long
- 14313 Like pattern string too long
- 14314 Username or password too long
- 14315 Illegal byte length of floating point number
- 14321 Illegal host variable type
- 14322 Illegal host variable type in like pattern string
- 14323 Username and password must be fixed length character strings
- 14331 The number of provided host variables does not match the number of parameters
- 14401 Column cannot be updated because it is not identified in the UPDATE clause of the SELECT statement of the cursor
- 14402 The table identified in the UPDATE or DELETE statement is not the same as that designated by the cursor
- 14403 Cannot describe statement without naming information
- 14404 Cannot update table <%> because the declare statement does not include a FOR UPDATE OF clause
- 14405 Cannot delete from table <%> because the declare statement is read-only

- 14406 Unexpected statement type encountered in an UPDATE or DELETE statement
- 14501 Database connection is not open
- 14601 Invalid cursor state
- 14611 Using-clause does not match dynamic parameters
- 14612 Using-clause does not match target specifications
- 14621 Cursor not found
- 14622 Ambiguous cursor name
- 14623 Invalid cursor name
- 14624 Cursor already allocated for statement
- 14631 Invalid SQL statement name
- 14641 Invalid SQL descriptor name
- 14642 Invalid descriptor index
- 14643 Invalid descriptor count
- 14651 Invalid exception number

C.9 Databank and table errors

These errors are associated with problems of physical access to databanks and tables. Locking errors should not result from transaction conflicts, but generally indicate either locking at the operating system level or malfunction of the internal MIMER routines. Many of the errors in this class are corrected by action taken at the operating system level. If errors persist in spite of corrective action, contact your MIMER representative.

- 16001 Table locked by another user
- 16002 Table locked by another cursor
- 16003 Foreign key table locked by another user
- 16004 Foreign key table locked by another cursor
- 16005 Log locked by another user
- 16006 Backup unit log file locked
- 16101 No databank <%> found in dictionary
- 16135 Record no longer exists
- 16141 Syntax error in filename for databank <%>
- 16142 Cannot open databank <%>, file <%> not found
- 16143 File protection violation for databank <%>, file <%>
- 16144 Cannot open databank <%>, file <%> locked by another user
- 16145 Too many databanks open concurrently (direct access I/O limit)
- 16146 File create error, disk full
- 16147 File create error for databank <%>, file <%>, quota exceeded
- 16148 Databank <%>, file <%>, device or network connection not ready
- 16149 Cannot open databank <%> in file <%>
- 16150 Tried to access databank <%> on node which is inaccessible because TRANSDB is OFFLINE
- 16151 Too many databanks open concurrently
- 16152 Tried to open non-MIMER databank

- 16154 Table control area exhausted
- 16155 Incompatible version of databank <%>
- 16156 Databank <%> belongs to another system databank
- 16157 Tried to open read-only databank with write access
- 16159 Old version of the databank <%> cannot be accessed without restoring the databank with the backup and restore utility
- 16160 Cannot set TRANSDB shadow OFFLINE on the same node as the master TRANSDB
- 16161 Databank <%> disk space exhausted
- 16162 Databank LOGDB disk space exhausted
- 16163 Databank TRANSDB disk space exhausted
- 16172 Databank <%> locked by another MIMER/DB user
- 16182 Databank <%> corrupt
- 16183 Bad parameter
- 16184 I/O error
- 16185 Internal databank identifier invalid
- 16186 Internal table identifier invalid
- 16187 Shadow <%> in file <%> has illegal sequence number
- 16189 Corrupt bitmap page
- 16190 Table root entry not found
- 16191 Exclusive access to databank required for attempted operation
- 16192 Load not allowed in databank with TRANS or LOG option
- 16193 TRANSDB and/or LOGDB not open
- 16194 Error occurred in transaction commit phase
- 16195 Internal inconsistency detected
- 16196 No end of table mark found for tableid
- 16197 Shadow <%> is already OFFLINE
- 16198 Shadow <%> is already ONLINE
- 16199 Result of bitmap page I/O undefined
- 16200 Result of index page I/O undefined
- 16201 Result of root page I/O undefined
- 16202 Result of data page I/O undefined
- 16203 Corrupt index page
- 16204 Corrupt root page
- 16205 Corrupt data page
- 16206 Write set corrupt
- 16207 Databank <%>, file <%>, table <%> has invalid record length
- 16208 Unable to open databank <%>. SHADOW licence required
- 16209 Unable to open databank <%>. NETIO licence required
- 16210 Cleanup control area exhausted
- 16211 Not properly closed, dbcheck initiated
- 16212 TRANSDB restart directory corrupt
- 16213 Error when closing databank file. Please consult multiuser log file
- 16214 Blockdata DKBLK1 missing
- 16215 Error accessing remote TRANSDB, node will not be accessed further

- 16216 Blocksize not supported
- 16217 Error when generating internal key
- 16218 Operation not allowed. Licensed number of users exceeded
- 16219 Too many multi systems started
- 16220 Unable to retrieve limit on number of allowed users
- 16221 Lost contact with peer
- 16222 Record length from update before is invalid

C.10 Miscellaneous errors

These errors arise from miscellaneous problems which do not fall into the other classes. If the corrective action is not indicated by the error description, contact your MIMER representative for help.

- 18001 Blockdata BLKDS2 not included
- 18002 Cannot log in, error in SYSDB initialization
- 18003 No privilege to open log file
- 18004 Databank LOGDB already opened by another MIMER/DB user. Could not be opened exclusively to drop log records
- 18005 Unknown language
- 18006 Language not properly installed
- 18007 Unable to fetch message text
- 18008 Restore in wrong sequence attempted
- 18009 Mismatching version of Embedded SQL and MIMER/DB
- 18010 Invalid log record found during restore operation
- 18011 Mismatching version of MIMER/DB and utilities
- 18012 The transformation of a TRANSDB shadow to master was interrupted before completion. Please login to UTIL to complete the transformation
- 18013 MIMER/DB started from SYSDB shadow. Transform SYSDB shadow to master with UTIL, or restart system from master SYSDB
- 18014 Alter shadow not allowed in SQL for system databanks. Use utility program instead.
- 18015 Open with hold is not possible when temporary tables are use for evaluation of the query
- 18016 Cursor could not be opened with hold as it is not prepared with hold
- 18017 With hold functionality not supported
- 18018 The network server version does not support scroll
- 18019 Bad parameters passed to DBAPI4
- 18020 Unknown information code = <%> used
- 18021 Only SELECT, INSERT, UPDATE, and DELETE operations (excluding where current forms) may be compiled together in a single statement
- 18101 Operation not allowed. SHADOW licence required
- 18102 Operation not allowed. MIMER/DB licence required
- 18103 Operation not allowed. MIMER/DB Level2 licence required
- 18104 Operation not allowed. MIMER/NET Client licence required

- 18105 Operation not allowed. MIMER/NET Server licence required
- 18106 Operation not allowed. MIMER/NET PC-Server licence required
- 18107 Operation not allowed. Beta test version of MIMER requires BETA licence.
- 18201 SYSDB cannot be backed up using CREATE BACKUP without an ONLINE shadow
- 18231 <%> records dropped from LOGDB
- 18232 Shadow <%> is OFFLINE
- 18233 Unable to access databank <%>, because it is OFFLINE
- 18234 Error <%> occurred when trying to access shadow <%>
- 18235 Error <%> occurred while trying to access databank <%>
- 18236 Statistics updated for <%> tables
- 18237 Databank <%> does not have LOG option
- 18238 <%> records copied to incremental backup
- 18239 Unable to CONNECT, because database is OFFLINE
- 18240 Unable to CONNECT, because database is OFFLINE and one connection already exists
- 18241 Unable to CONNECT, because SYSDB is OFFLINE
- 18242 Unable to CONNECT, because SYSDB is OFFLINE and one connection already exists
- 18243 Unable to access databank <%>, because database is OFFLINE
- 18500 Database <%> not found in <SQLHOSTS>
- 18501 Database <%> unknown on remote node
- 18502 Handshake message invalid, incompatible protocol
- 18503 Only remote databases are allowed, specify database which is not local
- 18504 The network server version is not compatible
- 18505 Local memory pool in network server exhausted (DSDYN3)
- 18506 In the current version only one local (and several remote) databases can be connected at a time
- 18507 Unknown connection name <%> specified
- 18508 Already connected, connection name <%>
- 18509 Database name <%> invalid, too long or contains invalid characters
- 18510 Connection name <%> invalid, too long or contains invalid characters
- 18512 Illegal reentrant call
- 18513 Use another TCP/IP port number
- 18514 Too deep address indirection
- 18521 Error opening <SQLHOSTS>, filename syntax error
- 18522 Error opening <SQLHOSTS>, file not found
- 18523 Error opening <SQLHOSTS>, file protection violation
- 18524 Error opening <SQLHOSTS>, file locked
- 18525 Error opening <SQLHOSTS>, too many opened files
- 18526 Error opening <SQLHOSTS>, file create error , disk space exhausted

- 18527 Error opening <SQLHOSTS>, other error (-7)
- 18528 Error opening <SQLHOSTS>, other error (-8)
- 18529 Error opening <SQLHOSTS>, other error (-9)
- 18530 Error opening <SQLHOSTS>, illegal access options
- 18550 Invalid network package format
- 18551 Unknown request code in network package (<%>)
- 18552 Network package longer than expected
- 18553 Internal data structures corrupt (DSNEE4)
- 18554 The UTILITY program does not have client/server support
- 18595 Network partner disconnected
- 18601 Could not connect to database <%>, unknown node '<%>'
- 18602 Could not connect to database <%>, unknown protocol '<%>'
- 18603 Could not connect to database <%>, unknown interface '<%>'
- 18604 Could not connect to database <%>, unknown service '<%>'
- 18605 Could not connect to database <%>, chosen protocol not supported on <%> '<%>'
- 18606 Could not connect to database <%>, network type not supported '<%>'
- 18607 Could not connect to database <%>, remote node is unreachable '<%>'
- 18608 Bad parameter NETID=<%> passed to network package
- 18609 Invalid parameter RECLLEN=<%> passed to network package
- 18610 Invalid parameter BUFFER=<%> passed to network package
- 18611 Too many concurrent network connections
- 18612 Connection refused '<%>'
- 18613 Unexpected network event '<%>'
- 18614 The underlying network protocol does not have enough capabilities '<%>'
- 18615 Network service busy '<%>'
- 18616 Local or remote system resources are insufficient '<%>'
- 18617 Connection timed out '<%>'
- 18618 Insufficient privileges for attempted network operation '<%>'
- 18619 Unexpected network error '<%>'
- 18620 Network operation would block (asynch mode)
- 18621 Could not load network library
- 18622 Required routines missing from network library
- 18901 Multi-user system not started
- 18902 MIMER logins are currently disabled, try again later
- 18903 Access denied to MIMER multi-user system
- 18904 Unable to attach to multi-user system, no response
- 18905 Operation not allowed. Licensed number of users exceeded
- 18920 Machine dependent error -18920, please refer to users guide for explanation
- 18921 Machine dependent error -18921, please refer to users guide for explanation

- 18922 Machine dependent error -18922, please refer to users guide for explanation
- 18923 Machine dependent error -18923, please refer to users guide for explanation
- 18924 Machine dependent error -18924, please refer to users guide for explanation
- 18925 Machine dependent error -18925, please refer to users guide for explanation
- 18926 Machine dependent error -18926, please refer to users guide for explanation
- 18927 Machine dependent error -18927, please refer to users guide for explanation
- 18928 Machine dependent error -18928, please refer to users guide for explanation
- 18929 Machine dependent error -18929, please refer to users guide for explanation

C.11 Internal errors

These errors arise from malfunction in internal MIMER routines. Contact your MIMER representative for help.

- 19001 Program level list corrupt
- 19002 No program level found
- 19003 Statement list corrupt
- 19004 Output parameter list corrupt
- 19005 Table list corrupt
- 19006 Unable to find log file, LOGDB corrupt
- 19007 Inconsistency detected when trying to update dictionary
- 19008 Unable to open table MIMER.INDEXCOL
- 19009 Dictionary table MIMER.IDENT corrupt
- 19010 Unable to extract correct information from MIMER.DATABANK
- 19011 Unable to extract correct information from MIMER.INDEX
- 19012 Could not find foreign key tableid (<*>) in MIMER.TABLEX
- 19013 No user MIMER_SC found in MIMER.IDENT
- 19014 No user PUBLIC found in MIMER.IDENT
- 19015 Sysid record in MIMER.OBJECT not found
- 19016 Time record in MIMER.OBJECT not found
- 19017 Invalid MAE program
- 19018 Invalid operation code <*> at PC=<*>
- 19019 Missing IMC instruction
- 19020 Invalid function code passed to instruction <*>
- 19021 No databank control block found for <*>
- 19022 Bad function code passed to CDBDS3
- 19023 Invalid pointer to naming structure
- 19024 Severity message program corrupt

- 19025 Invalid table descriptor
- 19026 Invalid table descriptor, log status invalid
- 19027 Base table must be opened before index tables
- 19028 Table root entry not found
- 19029 Unable to change position on write set because no mark is set
- 19030 Invalid length for allocation of program space
- 19031 Invalid table type
- 19032 No table control block found
- 19033 Cannot delete databank file outside transaction
- 19034 Bad function code passed to CRDDS2
- 19035 Invalid foreign or secondary index descriptor
- 19036 Error detected when closing table, hash chain corrupt
- 19037 Invalid internal table type encountered
- 19038 Write set inconsistency encountered
- 19039 Invalid internal statement identifier
- 19040 Invalid internal system identifier
- 19041 Invalid internal user identifier
- 19042 The static statement cannot be compiled because it is already identified with some other statement
- 19043 The statement cannot be prepared because it is already identified with a static statement
- 19044 Transaction control block chain corrupt
- 19045 Shadow <%> cannot be transformed because it is OFFLINE
- 19046 Databank <%> is referenced but not opened
- 19047 Table has not been opened with sufficient access to allow current operation
- 19048 Databank <%>, no shadow is found in dictionary with sequence number = <%>
- 19049 The internal update operation has not been prepared with the old record
- 19050 Incompatible version of data dictionary
- 19051 Compiled LIKE pattern corrupt
- 19052 Could not store lookup path, inconsistency detected
- 19053 Output descriptor overflow
- 19054 Unable to initialize database system
- 19055 Unable to generate a primary key
- 19056 Inconsistent user identifier (not logged in)
- 19057 Scroll program corrupted
- 19058 Extended name not supported in static SQL
- 19059 Invalid error message descriptor
- 19061 Loss of significance for VARCHAR length
- 19062 Positive overflow for VARCHAR length
- 19063 Negative overflow for VARCHAR length
- 19065 Bad parameter for VARCHAR length
- 19066 Illegal operand for VARCHAR length

- 19067 Bad record number
- 19068 No matching record
- 19071 Unrecognized data types for conversion
- 19072 Invalid read set record
- 19073 Insufficient internal descriptor area
- 19201 System error : <%> - Area outside MAE data storage
- 19202 System error : <%> - Attempt to qqwsal() in closed area
- 19203 System error : <%> - Cost value out of range
- 19204 System error : <%> - Error converting into Mimer format
- 19205 System error : <%> - Error from MDRCCI call
- 19206 System error : <%> - Error when reading databank option
- 19207 System error : <%> - Expression switch case not recognized
- 19208 System error : <%> - Factor was left unused
- 19209 System error : <%> - Failed to get a slave RST
- 19210 System error : <%> - Generation stack underflow
- 19211 System error : <%> - Group is not allocated
- 19212 System error : <%> - Host variable not defined
- 19213 System error : <%> - Host variable number mismatch
- 19214 System error : <%> - Illegal Set Func. mode switch case
- 19215 System error : <%> - Illegal Status switch case <%>
- 19216 System error : <%> - Index table not found
- 19217 System error : <%> - Invalid base pointer
- 19218 System error : <%> - Invalid object type
- 19219 System error : <%> - Invalid pointer
- 19220 System error : <%> - Main switch case not recognized
- 19221 System error : <%> - Multiple offset assignment
- 19222 System error : <%> - Multiple restriction groups
- 19223 System error : <%> - No area opened
- 19224 System error : <%> - Nonexistent member
- 19225 System error : <%> - No Tbl_desc for SCO
- 19226 System error : <%> - NOT stack overflow
- 19227 System error : <%> - NOT stack underflow
- 19228 System error : <%> - Offset outside MAE data storage
- 19229 System error : <%> - qqcbix() with illegal operator
- 19230 System error : <%> - qqcunx() with illegal operator
- 19231 System error : <%> - Error from MDRCFC call
- 19232 System error : <%> - qqrlst() with NULL list
- 19233 System error : <%> - qqwlst() with NULL list
- 19234 System error : <%> - Query result stack underflow
- 19235 System error : <%> - Query stack underflow
- 19236 System error : <%> - Rule matrix index out of range
- 19237 System error : <%> - Scan kind not implemented
- 19238 System error : <%> - Scan stack underflow
- 19239 System error : <%> - Selectivity factor value out of range
- 19240 System error : <%> - Semantic stack underflow

- 19241 System error : <%> - Set range violation
- 19242 System error : <%> - Set size incompatibility
- 19243 System error : <%> - Stack underflow
- 19244 System error : <%> - Statement switch case not recognized
- 19245 System error : <%> - Switch case not recognized
- 19246 System error : <%> - Too complicated UNION query
- 19247 System error : <%> - Too many nested subqueries
- 19248 System error : <%> - Traversal stack underflow
- 19249 System error : <%> - Unexpected EXPRESSION in HOST variables
- 19250 System error : <%> - Unexpected expression operand
- 19251 System error : <%> - Unexpected expression subtype
- 19252 System error : <%> - Unexpected node class
- 19253 System error : <%> - Unexpected SELECT ITEM
- 19254 System error : <%> - Unexpected statement subclass
- 19255 System error : <%> - Unexpected DD return code <%>
- 19256 System error : <%> - Unknown Host Variable type
- 19257 System error : <%> - Unknown statement type
- 19258 System error : <%> - WS stack overflow
- 19259 System error : <%> - X stack overflow
- 19260 System error : <%> - X stack underflow
- 19261 System error : <%> - Error logging is not enabled
- 19262 System error : <%> - Source position line or column is negative
- 19263 System error : <%> - Message insert string too long
- 19264 System error : <%> - Error logging is already enabled
- 19265 System error : <%> - MAE constant storage overflow
- 19266 System error : <%> - Selectivity rule number out of range
- 19267 System error : <%> - No entry for index id
- 19268 System error : <%> - qqcfnx() with illegal operator
- 19269 Internal error during INSERT
- 19270 System error : <%> - Scan queue underflow
- 19280 System error : <%> - Error from MDRTDC call
- 19901 Function not yet implemented

D DEPRECATED FEATURES

Some non-standard features in earlier versions of MIMER/SQL are deprecated, but retained for backward compatibility. Where these features have equivalents in the standard implementation, only the standard form is documented in the main body of this manual. Use of the standard forms is strongly recommended.

D.1 INCLUDE SQLCA

In earlier versions of MIMER/SQL use of `INCLUDE SQLCA` in an embedded SQL program was required to include the declaration of the SQL communication area. In MIMER/SQL V7.3 this is no longer required, instead applications should use the `SQLSTATE` variable and the `GET DIAGNOSTICS` statement to get all the information they got from `SQLCA` previously. See Chapter 3 for a description of `SQLSTATE` and `GET DIAGNOSTICS`. For compatibility reasons the use of `INCLUDE SQLCA` is still supported.

D.2 SQLCODE

The use of `SQLCODE` to retrieve status information has been replaced by the use of `SQLSTATE` and `GET DIAGNOSTICS` in version 7.3 of MIMER/SQL. For compatibility reasons, return codes can still be retrieved in `SQLCODE`, as in earlier versions of MIMER/SQL. However, in version 7.3 of MIMER/SQL `SQLCODE` can be either a field in the `SQLCA`, or a 4-byte integer variable in the application. MIMER will assume the existence of an `SQLCODE` variable in the application if no `INCLUDE SQLCA` statement is found and neither `SQLSTATE` nor `SQLCODE` is declared between `BEGIN DECLARE SECTION` and `END DECLARE SECTION`.

The values of `SQLCODE` are the same as the values for the internal MIMER return codes described in Appendix C.

D.3 SQLDA

The SQL descriptor area `SQLDA`, which was used in earlier version of MIMER/SQL, has now been replaced by a standardized SQL descriptor area. The `SQLDA` area was allocated and maintained by constructions in the host language. The new SQL descriptor area is allocated and maintained by standardized embedded SQL statements.

The former SQL descriptor area `SQLDA` is still supported in MIMER/SQL for compatibility reasons.

D.4 Parameter marker representation

In earlier versions of MIMER/SQL it was documented that a parameter marker in a dynamic SQL statement could be represented by either a question mark or a parameter name preceded by a colon. The non-standard representation of a parameter name preceded with a colon is now deprecated, but it is still supported for compatibility reasons.

D.5 VARCHAR(size)

In earlier versions of MIMER/SQL a VARCHAR structure was documented, which was used in the C language for handling variable-length character strings. This VARCHAR structure was defined as:

```
#define VARCHAR(x)
    struct {
        short    len;
        char     text[x];
    }
```

where the character string is stored in *text*, and the length of the text is stored in *len*.

E PHYSICAL IMPLEMENTATION

E.1 Databank handling

E.1.1 Databank-id's

Databanks are identified internally by a unique system identifier (databank-id) generated when the databank is created and stored in the data dictionary. The file containing a given databank is also marked with the databank-id. All operations which access the physical file for databank handling check that the id in the file is the same as that for the databank in the data dictionary, ensuring that the correct databank file is addressed.

This means that databank files cannot be simply moved or copied between different MIMER systems, even if the table definitions in the databank are the same in the two systems or the databank is empty. Databanks are moved between systems with the EXPORT/IMPORT utility (see the System Management Handbook), which ensures that the databank-id is correctly set for the system in which the databank will be used.

E.1.2 Databank files

The initial size of a databank is specified in MIMER-pages when the databank is created. A file of the corresponding size is created in the operating system.

More file space will automatically be allocated to the databank as necessary, in increments determined by the operating system.

Databanks can be extended with a specified number of pages using the ALTER DATABANK statement. In many situations, this procedure is recommended over automatic file extension. This is because space reserved for the databank file in a CREATE or ALTER DATABANK operation is generally contiguous (as far as possible) on the physical disk, while space allocated through several small automatic file extensions may well be scattered giving an unnecessarily fragmented physical file structure.

The name of the file in which a databank is stored is specified when the databank is created, and is stored in the data dictionary in exactly the form in which it is given in the CREATE DATABANK statement. In general, the operating system can provide default values where appropriate for physical disk, file directory or catalogue, and filename extension if these parameters are not specified. However, we recommend that a full file specification is always given in the CREATE DATABANK statement. This increases the documentary value of the contents of the data dictionary and reduces the risk for confusion. Logical names may be used in the file name if these are supported by the operating system.

The file name for a databank may be changed with the ALTER DATABANK statement. Note that this simply changes the entry in the data dictionary; the file must then be renamed in the operating system as a separate operation. The ALTER DATABANK statement cannot be used to redirect databank access operations to another, already existing databank file (unless it is a copy of the correct file) since the databank-id will not be correct.

Dropping a databank with the DROP DATABANK statement also deletes physical file in the operating system.

E.2 Table storage structure

Data in MIMER tables is stored in a B*-tree structure. Other physical storage structures are not supported.

As rows are inserted or deleted from tables, the tree structure is reorganized automatically. No periodical reorganizations are necessary. The algorithm for maintaining the structure gives an average of 83% (5/6) used space in each node.

E.3 Views

Views are implemented as logical entities, not as physical tables. A data manipulation request which addresses a view is mapped internally against the view definition to a request addressing the respective base tables. This means that certain operations may be permissible on base tables but syntactically incorrect for views, because the mapping process generates an illegal construction. For example, consider the view:

```
CREATE VIEW BILL_TOTAL (CUSTOMER, TOTAL)
AS SELECT RESERVATION, SUM(AMOUNT)
   FROM BILL
   GROUP BY RESERVATION
```

Certain retrieval operations on this view are illegal, such as

```
SELECT *
FROM BILL_TOTAL
WHERE TOTAL < 500
```

because the resolved query on the base table becomes

```
SELECT RESERVATION, SUM(AMOUNT)
FROM BILL
WHERE SUM(AMOUNT) < 500
GROUP BY RESERVATION
```

The search condition 'WHERE SUM(AMOUNT) < 500' is illegal.

If syntax errors are reported in apparently correct statements, investigate whether the table reference addressed is a view whose definition limits the permissible operations.

E.4 Indexes

Rows in a table are identified by the values of their primary keys, which consist of one or more of the columns in the table. The primary key of a table is also a primary index. Retrieval of a row for a given primary key is very efficient.

Secondary indexes are implemented by creating internal tables invisible to the user. The secondary index table is fully maintained by MIMER. Updating the original table results in automatic revision of the secondary index table. Thus while retrieval efficiency may be improved by the definition of a secondary index, the update time for the table is increased.

E.5 Compiled statements

SQL statements in application programs are compiled at run-time. If an identical statement is issued on two separate occasions in one program, it will be compiled separately for each occurrence. Statements repeated by means of loop constructions in the host program are however compiled only once. The performance of an application program may therefore be improved by using loops or equivalent constructions as far as possible for repeated statements.

E.6 Work space requirements

Many select statements (both `SELECT INTO` statements and cursor declarations) used in application programs require the construction of intermediate work tables. For instance data fetched from several tables, grouped with a `GROUP BY` clause, and sorted with an `ORDER BY` clause must for practical reasons be processed through intermediate physical storage. These work tables are placed in `SQLDB` and are invisible to the user.

Depending on the demands made in the `SELECT` statements, the amount of storage required in `SQLDB` may be considerable even if the final result table is relatively small. For this reason the size allocated to `SQLDB` when the system is installed should be generous, particularly in systems where automatic file extension is costly or prohibited. Chapter 6 of the System Management Handbook describes how to generate a new `SQLDB`.

Temporary work tables are constructed for cursors when the first `FETCH` operation is performed. The temporary tables are destroyed when the cursor is closed. In order to maintain consistency between the result set addressed by a cursor (represented in the temporary work table) and the contents of the database, work tables are not permitted to survive over the termination of a transaction. (All cursors are therefore closed by the statements `COMMIT` and `ROLLBACK`).

F APPLICATION PROGRAM EXAMPLES

This chapter illustrates the use of embedded SQL in application programs in each of the host languages supported by MIMER/SQL. The source code for the application example is supplied with the preprocessor for each language.

The last section of this chapter illustrates an example of how dynamic SQL can be used with MIMER/SQL 7.3. This example is only supplied in the C language.

F.1 C program example

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
** Example of X/Open CAE SQL (1992) compliant embedded program.
**
** The program tries to connect to the DEFAULT database as SYSADM
** with password SYSADM.
** It will then create a cursor to the standard view
** INFORMATION_SCHEMA.TABLES, and fetch all records from it.
**
** If any error occurs, the print_sqlerror routine will be called
** which prints an error message.
*/

/*
** X/Open does not allow the same variable to be defined in several
** DECLARE SECTIONS, so we declare SQLSTATE here so that all routines
** in the file can use the same variable.
*/
EXEC SQL BEGIN DECLARE SECTION;
char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;

void print_sqlerror()
/*
** print_sqlerror prints an error message for the latest error.
** Programmed according to the X/Open CAE specification
*/
{
/*
** Note: char x[YY] are fixed length and will be space padded to full ** length
** (see X/Open CAE and MIMER Programmers Reference Manual).
*/
EXEC SQL BEGIN DECLARE SECTION;
int i;
int exceptions;
char message[254];
int msglen;
EXEC SQL END DECLARE SECTION;

EXEC SQL GET DIAGNOSTICS :exceptions = NUMBER; /* How many exceptions? */
for (i=1; i<=exceptions; i++) {
EXEC SQL GET DIAGNOSTICS EXCEPTION :i
:message = MESSAGE_TEXT,
:msglen = MESSAGE_LENGTH;
message[msglen] = '\0';
printf("%s\n", message);
}
}

```

```
main()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char schema[30];
    char table[30];
    char type[16];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL DECLARE MYCURSOR CURSOR FOR
        SELECT TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE
        FROM INFORMATION_SCHEMA.TABLES;

    EXEC SQL WHENEVER SQLERROR goto error_exit;

    EXEC SQL CONNECT TO '' USER 'SYSADM' USING 'SYSADM'; /* Connect to DEFAULT */

    EXEC SQL OPEN MYCURSOR;

    while (1) {
        EXEC SQL FETCH MYCURSOR INTO :schema, :table, :type;
        if (strcmp(SQLSTATE, "02000") == 0) break; /* No more rows */
        printf("%s %s %s\n", schema, table, type);
    }

    EXEC SQL CLOSE MYCURSOR;

    EXEC SQL DISCONNECT DEFAULT;
    exit(0);

error_exit:
    print_sqlerror();

    EXEC SQL WHENEVER SQLERROR CONTINUE;

    EXEC SQL DISCONNECT DEFAULT;
    exit(0);
    return 0;
}
```

F.2 COBOL program example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
*
** Example of X/Open CAE SQL (1992) compliant embedded program.
**
** The program tries to connect to the DEFAULT database as SYSADM
** with password SYSADM.
** It will then create a cursor to the standard view
** INFORMATION_SCHEMA.TABLES, and fetch all records from it.
**
** If any error occurs, the print_sqlerror routine will be called which
** prints an error message.
*

DATA DIVISION.
WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 SQLSTATE PICTURE X(5).
    01 TABLE-SCHEMA PICTURE X(30).
    01 TABLE-NAME PICTURE X(30).
    01 TABLE-TYPE PICTURE X(16).

    01 EXCEPTIONS PIC S9(10) COMP.
    01 ERROR-MESSAGE PIC X(254).
    01 LINE-NUMBER PIC S9(10) COMP.
    EXEC SQL END DECLARE SECTION END-EXEC.

PROCEDURE DIVISION.
MAIN SECTION.

DO-IT.
    EXEC SQL DECLARE MYCURSOR CURSOR FOR
        SELECT TABLE-SCHEMA, TABLE-NAME, TABLE-TYPE
        FROM INFORMATION_SCHEMA.TABLES END-EXEC.

    EXEC SQL WHENEVER SQLERROR GO TO ERROR-EXIT END-EXEC.

** CONNECT TO DEFAULT DATABASE
EXEC SQL CONNECT TO '' USER 'SYSADM' USING 'SYSADM' END-EXEC.

EXEC SQL OPEN MYCURSOR END-EXEC.

PERFORM FETCH-AND-DISPLAY UNTIL SQLSTATE = "02000".

EXEC SQL CLOSE MYCURSOR END-EXEC.
EXEC SQL DISCONNECT DEFAULT END-EXEC.
STOP RUN.
```

```
FETCH-AND-DISPLAY.  
  EXEC SQL FETCH MYCURSOR INTO :TABLE-SCHEMA,  
                                :TABLE-NAME,  
                                :TABLE-TYPE      END-EXEC.  
  
  IF SQLSTATE NOT = "02000",  
    DISPLAY TABLE-SCHEMA, TABLE-NAME, TABLE-TYPE.  
  
ERROR-EXIT.  
  EXEC SQL WHENEVER SQLERROR CONTINUE  END-EXEC.  
  EXEC SQL GET DIAGNOSTICS :EXCEPTIONS = NUMBER END-EXEC.  
  PERFORM DISPLAY-ERROR-LINE VARYING LINE-NUMBER  
    FROM 1 BY 1  
    UNTIL LINE-NUMBER IS GREATER THAN EXCEPTIONS.  
  EXEC SQL DISCONNECT DEFAULT END-EXEC.  
  STOP RUN.  
  
DISPLAY-ERROR-LINE.  
  EXEC SQL GET DIAGNOSTICS EXCEPTION :LINE-NUMBER  
                                :ERROR-MESSAGE = MESSAGE_TEXT  
  END-EXEC.  
  
  DISPLAY ERROR-MESSAGE.  
  
END PROGRAM EXAMPLE.
```

F.3 FORTRAN program example

```

      PROGRAM SQLEX
C
C   Example of SQL-92 compliant embedded FORTRAN program.
C
C   The program tries to connect to the DEFAULT database as SYSADM
C   with password SYSADM.
C   It will then create a cursor to the standard view
C   INFORMATION_SCHEMA.TABLES, and fetch all records from it.
C
C   If any error occurs, the PSQLER routine will be called which
C   prints an error message.
C
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*5  SQLSTATE
      CHARACTER*30 SCHEMA
      CHARACTER*30 TABLE
      CHARACTER*16 TYPE
      EXEC SQL END DECLARE SECTION

      EXEC SQL DECLARE MYCURSOR CURSOR FOR
+      SELECT TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE
+      FROM INFORMATION_SCHEMA.TABLES

      EXEC SQL WHENEVER SQLERROR GOTO 9000

C   Connect to the default database
      EXEC SQL CONNECT TO '' USER 'SYSADM' USING 'SYSADM'

      EXEC SQL OPEN MYCURSOR

      EXEC SQL FETCH MYCURSOR INTO :SCHEMA, :TABLE, :TYPE
      DO WHILE (SQLSTATE .NE. '02000')
        WRITE(6,100) SCHEMA, TABLE, TYPE
100      FORMAT (X,A,X,A,X,A)
        EXEC SQL FETCH MYCURSOR INTO :SCHEMA, :TABLE, :TYPE
      END DO

      EXEC SQL CLOSE MYCURSOR

      GOTO 9999

9000 CONTINUE
      CALL PSQLER

9999 CONTINUE
      EXEC SQL WHENEVER SQLERROR CONTINUE
      EXEC SQL DISCONNECT DEFAULT
      END

```

```
      SUBROUTINE PSQLER
C
C   PSQLER prints an error message for the latest error.
C   Programmed according to the X/Open CAE specification
C
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*5  SQLSTATE
      INTEGER I,ERRORS,MSGLEN
      CHARACTER*254 MESSAGE
      EXEC SQL END DECLARE SECTION

      EXEC SQL GET DIAGNOSTICS :ERRORS = NUMBER
      DO 10 I=1,ERRORS
          EXEC SQL GET DIAGNOSTICS EXCEPTION :I
+           :MESSAGE = MESSAGE_TEXT,
+           :MSGLEN = MESSAGE_LENGTH
          WRITE (6,100) MESSAGE(:MSGLEN)
100     FORMAT(X,A)
10     CONTINUE

      RETURN
      END
```

F.4 Dynamic SQL program example

```

/*****
/*  dsqlsamp.c - sample program using dynamic sql driver
/*****
/*
/*          Simple command line oriented SQL executor
/*
/*****
/*  Created by Sysdeco Mimer AB.
/*
/*  You have a free right to use, modify, reproduce, and distribute the
/*  sample files (and/or any modified version) in any way you find
/*  useful, provided that you agree that Sysdeco Mimer has no warranty
/*  obligations or liability for any sample files which are modified.
/*
/*****

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "dsql.h"          /* function prototype definitions */

static  char  sqlbuf [4096];

static  void  error  ();
static  void  getid  (char* prompt,char* text);
static  char* savres (char* result,char* sqlbuf);
static  int*  savlen (int* collen,int colcount,char* sqlbuf);
static  void  show   (char* result,int rowcount,int* collen,int colcount);

/*****
int main()
/*****
{
    char  database [19];
    char  username [19];
    char  password [19];
    char* p;
    char* result;
    int*  collen;
    int   rc;
    int   cursor;
    int   colcount;
    int   rowcount;

    printf("\n      *** Welcome to Mimer Dynamic SQL Sample Program ***\n\n");
    /*
    *  Connect to database
    */
    for (;;)
    {
        getid("Database:",database);
        getid("Username:",username);
        getid("Password:",password); /* NOTE: password is not hidden! */
        printf(" \n");
        if ((rc = DsqlConnect(database,username,password)) > 0) break;
    }
}

```

```

        if (rc < 0) error(), exit(0);
        printf("You have entered wrong username or password.\n");
        printf("Please, try again.\n\n");
    }
    printf(" - End an SQL statement with ';' \n");
    printf(" - Exit the program by giving an empty command \n\n");
    /*
    * SQL command loop
    */
    for (;;)
    {
        /*
        * Read an SQL statement ending with ';'
        */
        printf("SQL>"); /* Display command prompt */
        for (p = sqlbuf; p++, *p++ = ' ')
        {
            *p = '\0';
            gets(p);
            p += strlen(p);
            while ((*p == '\0' || *p == ' ') && p > sqlbuf) p--;
            if (*sqlbuf == '\0' || *p == ';') break;
            printf(" >");
        }
        printf(" \n");
        if (*p == '\0') /* Empty command, ask if exit */
        {
            printf("Quit? (y/n) ");
            gets(p);
            printf(" \n");
            if (*p == 'y' || *p == 'Y') break;
            continue;
        }
        *p = '\0'; /* Eliminate the ending ';' */
        /*
        * Execute SQL-statement
        */
        cursor = DsqlExecute(sqlbuf);
        if (cursor > 0)
        {
            /*
            * It was a select statement, get column names
            */
            if ((colcount = DsqlColNames(cursor, sqlbuf, sizeof(sqlbuf))) > 0)
            {
                result = savres(NULL, sqlbuf);
                collen = savlen(NULL, colcount, sqlbuf);
                /*
                * Fetch the resulting rows into memory
                * and calculate the minimum possible width of the columns
                */
                rowcount = 0;
                while ((rc = DsqlFetch(cursor, sqlbuf, sizeof(sqlbuf))) > 0)
                {
                    result = savres(result, sqlbuf);
                    collen = savlen(collen, colcount, sqlbuf);
                    rowcount++;
                }
            }
        }
    }

```

```
        if (rc < 0 || DsqlClose(cursor) < 0) error();
        /*
         * Show the result set nicely formatted
         */
        show(result,rowcount,colllen,colcount);
    }
    else
    {
        error();
    }
}
else if (cursor == 0)
{
    printf("*** SQL statement executed successfully! ***\n\n");
}
else
{
    error();
}
}
/*
 * Disconnect and exit
 */
if (DsqlDisconnect() < 0) error();
printf("      *** Exit from Mimer Dynamic SQL Sample Program ***\n\n");
exit(0);
return 0;
}
```

```

/*****
/*
/* Print SQL error message
/*
/*
/*****
static void error()
/*****
{
    DsqlError(sqlbuf,sizeof(sqlbuf));
    printf("%s\n",sqlbuf);
}

/*****
/*
/* Prompt for connect identifiers
/*
/*
/*****
static void getid(char* prompt,char* text)
/*****
{
    printf("%s",prompt);
    sqlbuf[0] = '\0';
    gets(sqlbuf);
    sqlbuf[18] = '\0';
    strcpy(text,sqlbuf);
}

/*****
/*
/* Save a result row in memory
/*
/*
/*****
static char* savres(char* result,char* sqlbuf)
/*****
{
    static int reslen;
    int buflen = strlen(sqlbuf) + 1;

    if (result == NULL)
    {
        result = malloc(buflen + buflen - 1);
        memcpy(result,sqlbuf,buflen);
        reslen = buflen - 1;
    }
    else if((result = realloc(result,reslen + buflen)) == NULL)
    {
        printf("*** Fatal: not enough memory, exiting... **\n\n");
        exit(0);
    }
    memcpy(&result[reslen],sqlbuf,buflen);
    reslen += buflen - 1;
    return result;
}

```

```

/*****
/*
/* Keep track of the minimum possible width of the columns
/*
/*
/*****
static int* savlen(int* collen,int colcount,char* sqlbuf)
/*****
{
    char* p = strtok(sqlbuf,"\t\n");
    int i,n;

    if (collen == NULL) collen = calloc(sizeof(int),colcount);
    for (i = 0; i < colcount; i++)
    {
        n = p ? strlen(p) : 0;
        if (collen[i] < n) collen[i] = n;
        p = strtok(NULL,"\t\n");
    }
    return collen;
}

/*****
/*
/* Print the formatted result set and release memory
/*
/*
/*****
static void show(char* result, int rowcount, int* collen, int colcount)
/*****
{
    char* p = strtok(result,"\t\n");
    int i,n,pos;

    printf("*** %d row%s selected ***\n\n", rowcount, rowcount == 1 ? "" : "s");
    for (n = 0; n < rowcount + 2; n++)
    {
        pos = 0;
        memset(sqlbuf,' ',sizeof(sqlbuf));
        for (i = 0; i < colcount; i++)
        {
            if (p) memcpy(&sqlbuf[pos],p,strlen(p));
            if (n == 1) memset(&sqlbuf[pos], '=',collen[i]);
            p = strtok(NULL,"\t\n");
            pos += collen[i] + 1;
        }
        sqlbuf[pos] = '\0';
        printf("%s\n",sqlbuf);
    }
    printf("\n");
    free(result);
    free(collen);
}

```

```
/*  
*****  
*/  
/* dsql.h */  
/*  
*****  
*/  
  
#ifndef DSQL_H  
#define DSQL_H  
  
int DsqlConnect (char* database, char* userid, char* passwd);  
int DsqlDisconnect (void);  
int DsqlExecute (char* sql);  
int DsqlColNames (int stmt, char* buffer, int len);  
int DsqlFetch (int stmt, char* buffer, int len);  
int DsqlClose (int stmt);  
int DsqlError (char* message, int len);  
  
#endif
```

```

/*****
/* dsql.ec - dynamic sql driver */
/*****
/*
/* This package is a dynamic SQL sample with the following functions: */
/*
/* DsqlConnect - connect to the database */
/* DsqlDisconnect - disconnect from the database */
/* DsqlExecute - execute an SQL statement */
/* DsqlColNames - get the column names of a result table */
/* DsqlFetch - fetch the next row from a result table */
/* DsqlClose - close a cursor */
/* DsqlError - get error message text */
/*
/*****
/*
/* Created by Sysdeco Mimer AB. */
/*
/* You have a free right to use, modify, reproduce, and distribute the */
/* sample files (and/or any modified version) in any way you find */
/* useful, provided that you agree that Sysdeco Mimer has no warranty */
/* obligations or liability for any sample files which are modified. */
/*
/*****

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "dsql.h" /* function prototype definitions */

#define TRUE 1
#define FALSE 0

/*
 * VARCHAR datatype definition
 */
#define SQL_VARCHAR 12
/*
 * SQLSTATE return code definitions
 */
#define SQL_SUCCESS "00000"
#define SQL_INSUFFICIENT_ITEM_DESCRIPTOR_AREAS "01005"
#define SQL_NO_DATA "02000"
#define SQL_CONNECTION_REJECTED "28000"
/*
 * Shared data areas
 */
exec sql begin declare section;
static char statement [19]; /* Extended statement name */
static char cursor [19]; /* Extended cursor name */
static char descriptor [19]; /* SQL descriptor name */
static char sqlstate [6]; /* SQLSTATE */
static char buffer [8192]; /* Data buffer */
exec sql end declare section;

static int seqno = 0; /* Sequence number for generating unique names */

```

```

/*****
/*
/* Connect to the database.
/* Parameters:
/*
/* char* database : Database to be connected
/* char* username : User identification
/* char* password : Password
/*
/* Return codes:
/*
/* 1 : successful connection
/* 0 : wrong username or password
/* -1 : error (call DsqlError for message text)
*****/
int DsqlConnect(char* database,char* username,char* password)
/*****
{
    exec sql whenever sqlerror goto exception;
    exec sql begin declare section;
    char did [19] = "";
    char uid [19] = "";
    char pwd [19] = "";
    exec sql end declare section;
    char* p;

    /*
    * Check arguments.
    */
    if (database == NULL || username == NULL || password == NULL) return -1;
    /*
    * Connect to database.
    */
    strncpy(did,database,sizeof(did));
    did[sizeof(did) - 1] = '\0';
    strncpy(uid,username,sizeof(uid));
    uid[sizeof(uid) - 1] = '\0';
    strncpy(pwd,password,sizeof(pwd));
    pwd[sizeof(pwd) - 1] = '\0';

    for (p = uid; *p; p++) *p = (char)toupper((int)*p);

    exec sql connect to :did user :uid using :pwd;

    if (strcmp(sqlstate, SQL_CONNECTION_REJECTED) == 0) return 0;
    return 1;

exception:

    return -1;
}

```

```

/*****
/*
/* Disconnect from the database.
/*
/* Return codes:
/*
/*      0 : OK
/*     -1 : error (call DsqlError for message text)
/*
/*****
int DsqlDisconnect()
/*****
{
    exec sql whenever sqlerror goto exception;

    exec sql disconnect;
    return 0;

exception:

    return -1;
}

```

```

/*****
/*
/* Execute an SQL statement.
/*
/* Parameter:
/*
/* char* sqlstmt : SQL statement to be executed
/*
/* Return codes:
/*
/* +n : a select statement successfully opened,
/* returning a cursor identifier
/* 0 : a non-select statement executed successfully
/* -1 : error (call DsqlError for message text)
/*
/* Note:
/*
/* It is assumed that the SQL statement does not contain any
/* parameter markers. Unique statement, descriptor, and cursor names
/* are used, which make it possible to have several cursors open at
/* the same time.
/*
/*****
int DsqlExecute(char* sqlstmt)
/*****
{
    exec sql whenever sqlerror goto exception;
    exec sql begin declare section;
    int count;
    int n;
    int type;
    int length;
    exec sql end declare section;

    /*
    * Check argument.
    */
    if (sqlstmt == NULL) return -1;
    /*
    * Make unique statement and descriptor names.
    */
    seqno++;
    sprintf(statement, "%d",seqno);
    sprintf(descriptor,"%d",seqno);
    /*
    * Copy SQL statement to buffer.
    */
    strncpy(buffer,sqlstmt,sizeof(buffer));
    buffer[sizeof(buffer) - 1] = '\0';
    /*
    * Prepare SQL statement.
    */
    exec sql prepare :statement from :buffer;
    /*
    * Allocate descriptor.
    */
    exec sql allocate descriptor :descriptor;

```

```

/*
 * Describe output.
 */
exec sql describe output :statement
      using sql descriptor :descriptor;
if (strcmp(sqlstate, SQL_INSUFFICIENT_ITEM_DESCRIPTOR_AREAS) == 0)
{
  /*
   * The descriptor area was insufficient.
   * Allocate a new one with appropriate size.
   */
  exec sql get descriptor :descriptor :count = count;
  exec sql deallocate descriptor :descriptor;
  exec sql allocate descriptor :descriptor
        with max :count;
  exec sql describe output :statement
        using sql descriptor :descriptor;
}
/*
 * Get descriptor count to see if this was a select statement or not.
 */
exec sql get descriptor :descriptor :count = count;
if (count == 0)
{
  /*
   * Non-select statement. Execute.
   */
  exec sql execute :statement;
  /*
   * Deallocate statement and descriptor.
   */
  exec sql deallocate prepare :statement;
  exec sql deallocate descriptor :descriptor;
  /*
   * Return successful completion of non-select statement.
   */
  return 0;
}
else
{
  /*
   * Select statement. Set all columns to VARCHAR(512).
   * MIMER/SQL automatic type conversion will
   * handle numeric data.
   */
  type = SQL_VARCHAR;
  length = 512;
  for (n = 1; n <= count; n++)
  {
    exec sql set descriptor :descriptor value :n type = :type,
                                          length = :length;
  }
  /*
   * Make a unique cursor name.
   */
  sprintf(cursor, "%d", seqno);
  /*
   * Allocate and open cursor.
   */
  exec sql allocate :cursor cursor for :statement;
  exec sql open :cursor;
}
}

```

```
        /*
        * Return cursor identifier.
        */
        return seqno;
    }

exception:
    return -1;
}
```

```

/*****
/*
/* Get the column names of a result table.
/*
/* Parameters:
/*
/*     int    c      : cursor identifier returned from DsqlExecute
/*     char* record : buffer for the result string
/*     int    len    : length of the buffer (must be >= 32)
/*
/* Return codes:
/*
/*     +n : OK - number of columns
/*     -1 : error (call DsqlError for message text)
/*
/* Note:
/*
/*     The column names are returned in a tab ('\t') separated string
/*     ending with a newline ('\n'). If the string is too large, it is
/*     truncated.
/*
/*****
int DsqlColNames(int c,char* record,int len)
/*****
{
    exec sql whenever sqlerror goto exception;
    exec sql begin declare section;
    int count;
    int n;
    char name [19];
    exec sql end declare section;
    int length;

    /*
    * Check arguments.
    */
    if (record == NULL || len < 32) return -1;
    /*
    * Build descriptor name.
    */
    sprintf(descriptor,"%d",c);
    /*
    * Get number of columns.
    */
    exec sql get descriptor :descriptor :count = count;
    for (n = 1; n <= count; n++)
    {
        /*
        * Get one column name.
        */
        exec sql get descriptor :descriptor value :n
            :name = name;
        /*
        * Trim spaces and append a tab.
        */
        length = strlen(name);
        while (length > 1 && name[length - 1] == ' ') length--;
        if (length > len - 2) length = len - 2;
        memcpy(record,name,length);
        record += length;
        len -= length;
    }
}

```

```
        if (len == 2) break;
        if (n < count)
        {
            *record++ = '\t';
            len--;
        }
    }
    *record++ = '\n';
    *record = '\0';
    return count;

exception:

    return -1;
}
```

```

/*****
/*
/* Fetch the next row from a result table.
/*
/* Parameters:
/*
/*     int    c      : cursor identifier returned from DsqlExecute
/*     char* record : buffer for the result string
/*     int    len    : length of the buffer (must be >= 32)
/*
/* Return codes:
/*
/*     +n : OK - number of columns
/*     0  : No more data (End of File)
/*     -1 : error (call DsqlError for message text)
/*
/* Note:
/*
/*     The column values are returned in a tab ('\t') separated string
/*     ending with a newline ('\n'). It is assumed that the data does
/*     not contain tab, newline or null characters. A column value of
/*     NULL will be return as '?'. If the string is too large, it is
/*     truncated.
/*
/*****
int DsqlFetch(int c,char* record,int len)
/*****
{
    exec sql whenever sqlerror goto exception;
    exec sql begin declare section;
        int    count;
        int    length;
        int    isnull;
        int    n;
    exec sql end declare section;

    /*
    * Check arguments.
    */
    if (record == NULL || len < 32) return -1;
    /*
    * Build cursor and descriptor names.
    */
    sprintf(cursor,    "%d",c);
    sprintf(descriptor,"%d",c);
    /*
    * Fetch next row from result table.
    */
    exec sql fetch next from :cursor
        into sql descriptor :descriptor;
    if (strcmp(sqlstate,SQL_SUCCESS) == 0)
    {
        /*
        * Get number of columns.
        */
        exec sql get descriptor :descriptor :count = count;

```

```

for (n = 1; n <= count; n++)
{
    /*
     * Get one column value.
     */
    exec sql get descriptor :descriptor value :n :buffer = data,
                        :length = returned_length,
                        :isnull = indicator;

    /*
     * Check NULL indicator, trim spaces, append tab.
     */
    if (isnull == -1)
    {
        *record++ = '?';
        len--;
    }
    else
    {
        while (length > 1 && buffer[length - 1] == ' ') length--;
        if (length > len - 2) length = len - 2;
        memcpy(record,buffer,length);
        record += length;
        len -= length;
    }
    if (len == 2) break;
    if (n < count)
    {
        *record++ = '\t';
        len--;
    }
}
*record++ = '\n';
*record = '\0';
return count;
}
else if (strcmp(sqlstate,SQL_NO_DATA) == 0)
{
    /*
     * No more data.
     */
    return 0;
}

exception:

    return -1;
}

```

```

/*****
/*
/* Close a cursor.
/*
/* Parameter:
/*
/* int c : cursor identifier returned from DsqlExecute
/*
/* Return codes:
/*
/* 0 : OK
/* -1 : error (call DsqlError for message text)
/*
/*****
int DsqlClose(int c)
/*****
{
    exec sql whenever sqlerror goto exception;

    /*
    * Build cursor, statement and descriptor names.
    */
    sprintf(cursor, "%d",c);
    sprintf(statement, "%d",c);
    sprintf(descriptor,"%d",c);
    /*
    * Close cursor.
    */
    exec sql close :cursor;
    /*
    * Deallocate statement and descriptor.
    */
    exec sql deallocate prepare :statement;
    exec sql deallocate descriptor :descriptor;
    return 0;

exception:

    return -1;
}

```

```

/*****
/*
/* Get error message text.
/*
/* Parameters:
/*
/* char* message : buffer for the result string
/* int len : length of the buffer (must be >= 32)
/*
/* Return codes:
/*
/* 0 : OK
/* -1 : error (cannot get message text)
/*
/* Note:
/*
/* The message text is returned in a newline ('\n') separated
/* string. If the string is too large, it is truncated.
/*
*****/
int DsqlError(char* message,int len)
/*****
{
    exec sql whenever sqlerror goto exception;
    exec sql begin declare section;
    int count;
    int length;
    char state [6]; /* returned_sqlstate may not be stored sqlstate */
    int n;
    exec sql end declare section;
    char* p;

    /*
    * Check arguments.
    */
    if (message == NULL || len < 32) return -1;
    /*
    * Get number of errors.
    */
    exec sql get diagnostics :count = number;
    for (n = 1; n <= count; n++)
    {
        /*
        * Get diagnostics of an error.
        */
        exec sql get diagnostics exception :n :buffer = message_text,
            :length = message_length,
            :state = returned_sqlstate;

        /*
        * Prepare message area.
        */
        if (length > len - 18) length = len - 18;
        memcpy(message,buffer,length);
        p = strchr(message,':');
        if (p) *p = '\n';
        if (length > 64)
        {
            p = strchr(&message[64], ' ');
            if (p) *p = '\n';
        }
    }
}

```

```
        message += length;
        len      -= length;
        *message++ = '\n';
        /*
         * Add SQLSTATE code.
         */
        strcpy(message, "SQLSTATE:");
        message += 9;
        strcpy(message, state);
        message += 5;
        *message++ = '\n';
        *message++ = '\n';
        len -= 17;
        if (len < 32) break;
    }
    *message = '\0';
    return 0;

exception:

    strcpy(message, "cannot get diagnostics\n\n");
    return -1;
}
```


F.5 Dynamic SQL example for handling binary data

```

/*****
/* blobsamp.ec - sample program for put/get binary data */
/*****
/*
/* Created by Sysdeco Mimer AB.
/*
/* You have a free right to use, modify, reproduce, and distribute the
/* sample files (and/or any modified version) in any way you find
/* useful, provided that you agree that Sysdeco Mimer has no warranty
/* obligations or liability for any sample files which are modified.
/*
/*****
/* This sample program just shows how to read the file 'blob.r' and
/* insert the data into the table:
/*
/*      BLOB(ID int,SEQ int,DATA varchar(15000) not null,
/*          primary key(ID,SEQ))
/*
/* and then how to fetch the data from the table BLOB and compare it
/* with the file 'blob.r'.
/*
/*****

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define SQL_NO_DATA "02000"

#define SQL_BINARY_VARCHAR -13 /* MIMER-specific type code */

exec sql begin declare section;
static char buffer[15001]; /* data buffer */
static char sqlstate[6];
static char select[35] = "select data from blob where id = ?";
static char insert[31] = "insert into blob values(?,?,?)";
static char delete[17] = "delete from blob";
exec sql end declare section;

static int putblob (int,char*);
static int chkblob (int,char*);
static int error (char*,int);

/*****
int main(void)
/*****
{
    exec sql whenever sqlerror goto exception;

    exec sql connect to '' user 'BLOB' using 'BLOB';

    exec sql execute immediate :delete; /* clear the table */

    if (putblob(1,"blob.r") < 0) return -1;
    if (chkblob(1,"blob.r") < 0) return -1;

    exec sql disconnect;

```

```
        return 0;

exception:

    error(buffer, sizeof(buffer));
    printf("%s\n", buffer);
    return -1;
}
```

```

/*****
/*
/* Put the contents of a file into the database
/*
/* Parameters:
/*
/* int blobid : identifier for the data
/* char* filename : name of the file
/*
/* Return codes:
/*
/* 0 : OK
/* -1 : error
/*
/*****
static int putblob(int blobid,char* filename)
/*****
{
    exec sql whenever sqlerror goto exception;
    exec sql begin declare section;
    int id;
    int seq;
    int type;
    int length;
    exec sql end declare section;
    FILE* file;

    if ((file = fopen(filename,"rb")) == NULL)
    {
        printf("open error\n");
        return -1;
    }
    id = blobid;
    type = SQL_BINARY_VARCHAR;

    exec sql prepare 'statement' from :insert;
    exec sql allocate descriptor 'input' with max 3;
    exec sql describe input 'statement' using sql descriptor 'input';

    for (seq = 1; !feof(file); seq++)
    {
        length = fread(buffer,1,15000,file);
        if (ferror(file))
        {
            printf("read error\n");
            return -1;
        }

        exec sql set descriptor 'input' value 1 data = :id;
        exec sql set descriptor 'input' value 2 data = :seq;
        exec sql set descriptor 'input' value 3 data = :buffer,
            type = :type,
            length = :length;

        exec sql execute 'statement' using sql descriptor 'input';
    }
    exec sql deallocate descriptor 'input';
    exec sql deallocate prepare 'statement';
    fclose(file);
    return 0;
}

```

```
exception:  
  
    error(buffer, sizeof(buffer));  
    printf("%s\n", buffer);  
    return -1;  
}
```

```

/*****
/*
/* Check that stored data are exactly the same as the file contents.
/*
/* Parameters:
/*
/*      int   blobid   : identifier for the data
/*      char* filename : name of the file
/*
/* Return codes:
/*
/*      0 : OK
/*      -1 : error
/*
/*****
static int chkblob(int blobid,char* filename)
/*****
{
    exec sql whenever sqlerror goto exception;
    exec sql begin declare section;
    int   id;
    int   type;
    int   length;
    exec sql end declare section;
    FILE* file;
    char  record[15000];

    if ((file = fopen(filename,"rb")) == NULL)
    {
        printf("open error\n");
        return -1;
    }
    id   = blobid;
    type = SQL_BINARY_VARCHAR;

    exec sql prepare 'statement' from :select;

    exec sql allocate descriptor 'input'  with max 1;
    exec sql allocate descriptor 'output' with max 1;

    exec sql describe input  'statement' using sql descriptor 'input';
    exec sql describe output 'statement' using sql descriptor 'output';

    exec sql set descriptor 'input'  value 1 data = :id;

    exec sql allocate 'cursor' cursor for 'statement';
    exec sql open 'cursor' using sql descriptor 'input';
    for (;;)
    {
        exec sql fetch next from 'cursor' into sql descriptor 'output';

        if (strcmp(sqlstate,SQL_NO_DATA) == 0) break;

        exec sql get descriptor 'output' value 1 :buffer = data,
                                                    :type   = type,
                                                    :length = returned_length;
    }
}

```

```
        if (fread(record,1,15000,file) != length ||
            memcmp(buffer,record,length) != 0)
        {
            printf("compare error\n");
            return -1;
        }
    }
    exec sql close 'cursor';
    exec sql deallocate descriptor 'output';
    exec sql deallocate descriptor 'input';
    exec sql deallocate prepare 'statement';
    fclose(file);
    printf("compare ok!\n");
    return 0;

exception:

    error(buffer,sizeof(buffer));
    printf("%s\n",buffer);
    return -1;
}
```

```

/*****
/*
/* Get error message text.
/*
/* Parameters:
/*
/* char* message : buffer for the result string
/* int len : length of the buffer (must be >= 32)
/*
/* Return codes:
/*
/* 0 : OK
/* -1 : error (cannot get message text)
/*
/* Note:
/*
/* The message text is returned in a newline ('\n') separated
/* string. If the string is too large, it is truncated.
/*
/*****
static int error(char* message,int len)
/*****
{
    exec sql whenever sqlerror goto exception;
    exec sql begin declare section;
    int count;
    int length;
    char state [6]; /* returned_sqlstate may not be stored in sqlstate */
    int n;
    exec sql end declare section;
    char* p;

    /*
    * Check arguments.
    */
    if (message == NULL || len < 32) return -1;
    /*
    * Get number of errors.
    */
    exec sql get diagnostics :count = number;
    for (n = 1; n <= count; n++)
    {
        /*
        * Get diagnostics of an error.
        */
        exec sql get diagnostics exception :n :buffer = message_text,
            :length = message_length,
            :state = returned_sqlstate;

        /*
        * Prepare message area.
        */
        if (length > len - 18) length = len - 18;
        memcpy(message,buffer,length);
        p = strchr(message,':');
        if (p) *p = '\n';
        if (length > 78)
        {
            p = strchr(&message[78], ' ');
            if (p) *p = '\n';
        }
    }
}

```

```
message += length;
len      -= length;
*message++ = '\n';
/*
 * Add SQLSTATE code.
 */
strcpy(message,"SQLSTATE:");
message += 9;
strcpy(message,state);
message += 5;
*message++ = '\n';
*message++ = '\n';
len -= 17;
if (len < 32) break;
}
*message = '\0';
return 0;

exception:

strcpy(message,"cannot get diagnostics\n\n");
return -1;
}
```

INDEX

A

- access rights 4-3
 - for cursors 5-4
- accessing data 5-1
- active connection 4-6
- ALLOCATE CURSOR 7-11
- ALLOCATE DESCRIPTOR 7-4
- arrays 3-1

B

- back-up protection 6-9
- BEGIN DECLARE SECTION 3-1

C

- C (programming language)
 - comments A-2
 - data types A-4
 - host variables A-3
 - line continuation A-2
 - null terminated strings A-3
 - preprocessor output A-5
 - see also host languages 2-2
 - statement delimiters A-2
- closing cursors 5-4
- COBOL
 - comments A-7
 - data types A-10
 - decimal point character A-7
 - host variables A-8
 - included code A-7
 - line continuation A-6
 - preprocessor output A-11
 - preprocessor switches B-2
 - quotation marks A-7
 - see also host languages 2-2
 - SQLCA declaration A-8
 - statement delimiters A-6
 - statement margins A-6
 - string delimiters A-7
- comments 2-3
 - in C A-2
 - in COBOL A-7
 - in FORTRAN A-14
- COMMIT 6-5
- committing transactions 6-1
- compiler 2-4
- concurrency control 6-1

- connecting to a database 4-4
- connection name 4-4
- connections
 - and transactions 6-5
 - cursors 5-4
- current row 5-8
- cursor-independent data manipulation 5-8
- cursors 5-1
 - access rights 5-4
 - and program ids 4-6
 - closing 5-4
 - declaration 5-2
 - evaluating SELECT statement 5-2
 - for join conditions 5-6
 - for UPDATE and DELETE 5-8
 - in dynamic SQL 7-15
 - in multiple connections 5-4
 - in transactions 6-8
 - opening 5-2
 - position in result set 5-3, 5-9
 - resource allocation 5-4
 - stacking 5-6

D

- data errors C-7
- data types
 - in C A-4
 - in COBOL A-10
 - in FORTRAN A-16
- databank access errors C-17
- databank files E-1
- databank id E-1
- database concept 4-1
- database connection 4-4
- database name 4-4
- deadlock 6-3
- DEALLOCATE DESCRIPTOR 7-4
- DEALLOCATE PREPARE 7-11
- decimal point character
 - in COBOL A-7
- declaration of SQLSTATE 3-3
- DECLARE SECTION 3-1
- declaring cursors 5-2
- declaring host variables 5-1
- DELETE 5-8
- delimiters
 - in C A-2
- deprecated features D-1
 - INCLUDE SQLCA D-1
 - parameter markers D-2
 - SQLCODE D-1
 - SQLDA D-1
 - VARCHAR(size) D-2
- DESCRIBE INPUT 7-13
- DESCRIBE OUTPUT 7-13
- DESCRIBE select-lists 7-13
- describing dynamic SQL statements 7-12
- designing transactions 6-2
- diagnostics area 3-4

- diagnostics area
 - information items 8-4
- disconnecting 4-5
- disk crash 6-10
- dormant cursors 5-4
- dynamic SQL 7-1
 - cursors 7-15
 - describing statements 7-12
 - example framework 7-16
 - executing statements 7-14
 - preparing statements 7-10
 - statement source form 7-10
 - statement object form 7-10
 - statements 7-3
- dynamic SQL program example F-1

E

- embedded SQL
 - see ESQL 2-1
- END DECLARE SECTION 3-1
- ending transactions 6-5
- ENTER 4-6
- error handling 8-1
 - in transactions 6-9
- ESQL
 - program structure 2-4
 - restrictions on host language code 2-3
 - scope 2-1
- exception conditions 8-1
- EXEC SQL 2-2
- executing dynamic SQL statements 7-14
- extended dynamic cursors 7-11
- extended dynamic statements 7-11

F

- FETCH 5-3
- FORTRAN
 - comments A-14
 - host variables A-15
 - included code A-14
 - line continuation A-13
 - preprocessor output A-17
 - SAVE statements B-2
 - see also host languages 2-2
 - SQLCA declaration A-14
 - statement delimiters A-13
 - statement margins A-13
 - statement numbers A-13

G

- GET DESCRIPTOR 7-7
- GET DIAGNOSTICS 3-4, 8-3
- GRANT OPTION 4-3
- group idents 4-2

H

- host languages 2-2, A-1
- host variables 3-2
 - arrays 3-1
 - declaration 3-1, 5-1

- declarations 2-4
 - in C A-3
 - in COBOL A-8
 - in cursor declarations 5-2
 - in dynamic SQL 7-2
 - in FORTRAN A-15
 - in SQL statements 3-1
- names 2-3

I

- identifying SQL statements 2-2
- idents 4-1
- IF A-14
- implicit connection 4-5
- included code
 - in COBOL A-7
 - in FORTRAN A-14
- indexes E-3
- indicator variables 3-2
- INSERT 5-8
- interactive environments 7-1
- internal MIMER return codes C-4
- interrupted transactions 6-10
- item descriptor area 7-5

J

- join retrievals
 - using cursors 5-6

L

- LEAVE 4-6
- LEAVE RETAIN 4-6
- limits C-8
- line continuation
 - in C A-2
 - in COBOL A-6
 - in FORTRAN A-13
- locking 6-3
- logging 6-9
- logical IF in FORTRAN A-14
- loops 2-4, 6-2

M

- minus sign
 - in COBOL variable names A-7
- multiple connections 4-5
- multiple tables in data retrieval 5-5

N

- non-SELECT statements in dynamic SQL 7-14, 7-15
- NULL 3-2
- null terminated strings A-3

O

- object form of dynamic SQL statements 7-10
- OPEN 5-2
- optimistic concurrency control 6-1
- optimization 2-4
- orientation specification 5-3
- OS_USER idents 4-1

P

- parameter markers 7-2
 - in SELECT statements 7-15
- Parts explosion 5-6
- positioning cursors 5-3
- preparing dynamic SQL statements 7-10
- preprocessing 2-3
 - WHENEVER statements 8-6
- preprocessor
 - input files B-1
- preprocessor output B-1
 - in C A-5
 - in COBOL A-11
 - in FORTRAN A-17
- preprocessor switches B-2
- privileges 4-3
- program construction errors C-15
- program examples F-1
- program idents 4-2, 4-6
- program structure 2-4

Q

- quotation marks
 - in C A-2
 - in COBOL A-7

R

- read-only cursors 5-2, 5-9
- read-set 6-1
- read-through-write-set 6-6
- retrieving data 5-3
- retrieving single rows 5-5
- ROLLBACK 6-5
- run-time errors 8-2

S

- SAVE statements in FORTRAN B-2
- scope of embedded SQL 2-1
- SCROLL 5-2
- scrollable cursor 5-2, 5-3
- SELECT INTO 5-5
- SELECT statement 5-1
- semantic errors 8-1
- SET DESCRIPTOR 7-7
- SET TRANSACTION CHANGES 6-6
- single row SELECT 5-5
- source form of dynamic SQL statements 7-10
- SQL compiler 2-4
- SQL descriptor area 3-4, 7-4
 - COUNT field 7-4
 - item descriptor area 7-5
 - structure 7-4
 - TYPE field 7-8
- SQL statement identifier 2-2
- SQL statements
 - compiling E-3
 - errors C-9
 - in loops 2-4
 - using host variables 3-1

- SQLCA
 - declaration in COBOL A-8
 - declaration in FORTRAN A-14
 - sqlhosts 4-4
 - SQLSTATE 3-3, 8-2, C-2
 - class 3-3
 - fields 3-3
 - return codes C-2
 - subclass 3-3
 - stacking cursors 5-6
 - starting transactions 6-4
 - statement delimiters
 - in C A-2
 - in COBOL A-6
 - in FORTRAN A-13
 - statement margins
 - in COBOL A-6
 - in FORTRAN A-13
 - statement numbers in FORTRAN: A-13
 - string delimiters
 - in COBOL A-7
 - subprogram names 2-3
 - subroutine names 2-3
 - syntax errors 8-1
 - system failure 6-10
- T**
- table access errors C-17
 - transactions 6-1
 - and cursors 6-8
 - build-up 6-1
 - design 6-2
 - error handling 6-9
 - interrupted 6-10
 - with loops 6-2
 - tree structure
 - traversing 5-6
 - types A-16
- U**
- updatable cursors 5-9
 - UPDATE 5-8
 - UPDATE CURRENT 5-8
 - user idents 4-1
 - utility commands 2-1
- V**
- variables
 - see host variables 2-4
 - view implementation E-3
- W**
- warnings C-4
 - WHENEVER 8-6
 - in transactions 6-9
 - white-space A-2, A-7, A-13
 - work tables E-4
 - write-set 6-1